보안을 위한 프로그램 분석 기법

최광훈

전남대학교

November 26, 2025

Contents

Chapter 1

구문 분석: 어휘 분석

Andrew W. Appel, Modern Compiler Implementation in ML, Cambridge University Press (Chapter 2)

1.1 어휘 분석(Lexical analysis)

> lex-i-cal: 언어의 문법이나 구조와 구별되는,
> 단어 혹은 어휘에 관한 것.
>
> *웹스터 사전(Webster's Dictionary)*

한 언어의 프로그램을 다른 언어로 번역하려면, 컴파일러는 먼저 프로그램을 분해하여 그 구조와 의미를 이해한 뒤, 다른 방식으로 다시 조립해야 한다. 컴파일러의 전단부(front end)는 **분석(analysis)**을 수행하고, 후단부(back end)는 **합성(synthesis)**을 수행한다.

분석 과정은 보통 다음과 같이 나뉜다.

- **어휘 분석(Lexical analysis)**: 입력을 개별 단어 또는 "토큰"으로 분해한다.
- **구문 분석(Syntax analysis)**: 프로그램의 구문 구조를 파악한다.
- **의미 분석(Semantic analysis)**: 프로그램의 의미를 헤아린다.

어휘 분석기는 문자 흐름(a stream of characters)을 받아 이름(name), 키워드(keyword), 구두점(punctuation mark)의 흐름(a stream of tokens)을 만들어낸다. 이 과정에서 토큰 사이의 공백(white space)과 주석(comments)은 버린다. 파서(parser)가 가능한 모든 지점에서 공백과 주석을 처리해야 한다면 지나치게 복잡해질 것이므로, 어휘 분석을 구문 분석과 분리하는 주요 이유가 여기에 있다.

어휘 분석 자체는 매우 복잡하지 않지만, 우리는 이를 높은 수준의 형식주의(formalism)[1]와 도구(tools)로 다룰 것이다. 왜냐하면 이와 유사한 형식주의는 구문 분석 연구에도 유용하고, 이와 유사한 도구들은 컴파일 이외의 많은 분야에서도 활용되기 때문이다.

[1] 형식: 수학이나 논리 규칙에 따라 예외 없이 표현하는 것

1.1.1 어휘 토큰(Lexical Tokens)

어휘 토큰(lexical token)은 프로그래밍 언어의 문법에서 하나의 단위로 취급될 수 있는 문자들의 연속이다. 프로그래밍 언어는 어휘 토큰을 유한 개의 토큰 타입으로 구성된 집합으로 분류한다. 예를 들어, 전형적인 프로그래밍 언어의 몇 가지 토큰 타입은 다음과 같다.

Type	**Examples**
ID	foo, nl4, last
NUM	73, 0, 00, 515, 082
REAL	66.1, .5, 10., 1e67, 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(
RPAREN	)

알파벳 문자로 구성된 `IF`, `VOID`, `RETURN` 같은 구두점(punctuation) 토큰은 예약어(reserved words)라고 불리며, 대부분의 언어에서 식별자로 사용할 수 없다.

토큰이 아닌 예시는 다음과 같다.

- 주석: `/* try again */`
- 전처리 지시어: `#include<stdio.h>`
- 전처리 지시어: `#define NUMS 5 , 6`
- 매크로: `NUMS`
- 공백, 탭, 개행 문자

매크로 전처리기가 필요한 언어에서는 전처리기가 소스 문자 스트림을 처리하여 또 다른 문자 스트림을 만들어내고, 이것이 어휘 분석기로 전달된다. 또한 매크로 처리를 어휘 분석과 통합하는 것도 가능하다.

예를 들어 다음과 같은 프로그램이 있다고 하자.

```
float match0(char *s) /* find a zero */
{ if (!strncmp(s, "0.0", 3))
    return 0.;
}
```

어휘 분석기는 이를 다음과 같은 토큰 스트림으로 반환한다.

```
FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN LBRACE IF LPAREN BANG ID(strncmp)
LPAREN ID(s) COMMA STRING(0.0) COMMA NUM(3) RPAREN RPAREN RETURN REAL(0.0) SEMI RBRACE EOF
```

각 토큰의 토큰 타입이 보고되며, 식별자(identifier)와 리터럴(literal) 같은 일부 토큰은 의미값(semantic value)을 함께 가진다. 이는 토큰 타입과 함께 추가적인 보조 정보로 사용된다.

그렇다면 프로그래밍 언어의 어휘 규칙은 어떻게 기술해야 할까? 어휘 분석기는 어떤 언어로 작성해야 할까?

우리말로 언어의 어휘 토큰을 설명할 수 있다. 예를 들어 C나 Java에서의 식별자는 다음과 같이 정의된다.

- 식별자는 문자(letter)와 숫자(digit)로 이루어진 연속이며, 첫 번째 문자는 반드시 문자여야 한다.
- 밑줄 _ 은 문자로 간주된다.
- 대문자와 소문자는 구분된다.
- 입력 스트림이 특정 문자까지 토큰으로 파싱되었다면, 다음 토큰은 가능한 가장 긴 문자열을 포함하는 토큰으로 선택한다.
- 공백, 탭, 개행, 주석은 토큰을 구분할 때를 제외하고는 무시된다.
- 인접한 식별자, 키워드, 상수를 구분하기 위해 일부 공백이 필요하다.

사실상 어떤 프로그래밍 언어를 사용해서도 임시(ad hoc) 어휘 분석기를 구현할 수 있다. 그러나 우리는 *정규 표현식(regular expressions)*이라는 형식 언어로 어휘 토큰을 명세하고, *결정적 유한 오토마타(DFA)*로 어휘 분석기를 구현하며, 수학을 통해 이 둘을 연결할 것이다. 이러한 접근은 더 단순하고 읽기 쉬운 어휘 분석기를 만드는 길로 이어진다.

1.2 정규식(Regular Expressions)

언어(language)란 문자열(string)들의 집합이라고 하자. 문자열은 유한한 기호(symbol)들의 연속이다. 기호 자체는 유한한 알파벳 집합에서 취해진다.

예를 들어,

- Pascal 언어는 합법적인 Pascal 프로그램을 구성하는 모든 문자열의 집합이다.
- 소수(primes)의 언어는 소수를 표현하는 모든 10진수 문자열의 집합이다.
- C 예약어 언어는 C 프로그래밍 언어에서 식별자로 사용할 수 없는 모든 알파벳 문자열의 집합이다.

앞의 두 언어는 무한 집합이고, 마지막 것은 유한 집합이다. 이 모든 경우에서 알파벳은 ASCII 문자 집합이다.

이러한 방식으로 언어를 말할 때, 우리는 문자열에 어떠한 의미도 부여하지 않는다. 단지 각 문자열이 그 언어에 속하는지 아닌지를 분류하려는 것뿐이다.

이들 언어를 유한한 기술로 명세하기 위해 **정규 표현식(regular expressions)** 표기를 사용한다. 각 정규 표현식은 문자열 집합을 나타낸다.

기호(Symbol): 알파벳 기호 **a**에 대해, 정규 표현식 **a**는 문자열 a만을 포함하는 언어를 의미한다.

선택(Alternation): 두 정규 표현식 M과 N이 주어졌을 때, 수직선(|)으로 쓰이는 선택 연산자는 새로운 정규 표현식 $M|N$을 만든다. 문자열이 $M|N$의 언어에 속한다는 것은, 그 문자열이 M의 언어에 속하거나 N의 언어에 속한다는 뜻이다. 따라서 **a|b**의 언어는 문자열 a와 b를 포함한다.

연결(Concatenation): 두 정규 표현식 M과 N이 주어졌을 때, 연결 연산자 $\cdot$은 새로운 정규 표현식 $M \cdot N$을 만든다. 문자열이 $M \cdot N$의 언어에 속한다는 것은, 문자열 α가 M의 언어에 속하고 문자열 β가 N의 언어에 속할 때, $\alpha\beta$의 형태로 이루어진다는 뜻이다. 예를 들어, 정규 표현식 $(a|b) \cdot a$는 문자열 aa와 ba를 포함하는 언어를 정의한다.

엡실론(Epsilon): 정규 표현식 ϵ은 오직 공백 문자열만을 포함하는 언어를 의미한다. 따라서 $(a \cdot b)|\epsilon$은 {“”, “ab”}를 포함하는 언어를 나타낸다.

반복(Repetition): 정규 표현식 M에 대해, 그 클레이니 폐포(Kleene closure) M^*는 M의 문자열들을 0번 이상 이어 붙여 만든 모든 문자열의 집합을 의미한다. 예를 들어, $((a|b) \cdot a)^*$는 무한 집합

$$\{\text{“”}, \text{“}aa\text{”}, \text{“}ba\text{”}, \text{“}aaaa\text{”}, \text{“}baaa\text{”}, \text{“}aaba\text{”}, \text{“}baba\text{”}, \text{“}aaaaaa\text{”}, \dots\}$$

를 나타낸다.

기호(symbol), 선택(alternation), 연결(concatenation), 엡실론(epsilon), 그리고 클레이니 폐포(Kleene closure)를 사용하여 프로그래밍 언어의 어휘 토큰에 해당하는 ASCII 문자 집합을 명세할 수 있다. 먼저, 몇 가지 예를 살펴보자.

- $(0|1)^* \cdot 0$: 2의 배수인 이진수 문자열.
- $b^*(abb^*)^*(a|\epsilon)$: 연속된 a가 없는 a, b 문자열.
- $(a|b)^*aa(a|b)^*$: 연속된 a를 포함하는 a, b 문자열.

정규 표현식에서 $\cdot$ 나 ϵ은 종종 생략되며, 우선순위는 클레이니 폐포 > 연결 > 선택이다. 따라서 $ab|c$는 $(a \cdot b)|c$를 의미하고, $(a|)$는 $(a|\epsilon)$을 의미한다.

표현식	**설명**
a	문자 a 자체
ϵ	공백 문자열
$M\|N$	선택 (둘 중 하나)
$M \cdot N$ 또는 MN	연결 (앞뒤로 이어 붙임)
M^*	반복 (0회 이상)
M^+	반복 (1회 이상)
$M?$	선택적 (0회 또는 1회)
$[a-zA-Z]$	문자 집합 중 하나
.	임의의 문자 (개행 제외)
”abc”	따옴표 안 문자열 그대로

Figure 1.1: 정규 표현식 표기법

더 많은 축약 표기법을 소개해 보자. `[abcd]`는 $(a|b|c|d)$를 의미하고, `[b-g]`는 `[bcdefg]`를 의미한다. `[b-gM-Qkr]`는 `[bcdefgMNOPQkr]`를 의미한다. $M?$는 $(M|\epsilon)$을 의미하고, M^+는 $(M \cdot M^*)$를 의미한다.

정규식	토큰 타입
`if`	(IF);
`[a-z][a-z0-9]*`	(ID);
`[0-9]+`	(NUM);
`([0-9]+"."[0-9]*) | ([0-9]*"."[0-9]+)`	(REAL);
`"--"[a-z]*"\n" | (" "|\n|\t")+`	(continue());
`.`	(error(); continue());

Figure 1.2: 일부 토큰에 대한 정규 표현식 (원문 유지)

이러한 확장 표기들은 편리하지만 정규 표현식의 표현 능력을 확장하지는 않는다. 즉, 이 축약으로 표현할 수 있는 문자열 집합은 모두 기본 연산자들만으로도 표현할 수 있다. 모든 연산자는 Figure 1.1에 요약되어 있다.

이 언어를 사용하면 프로그래밍 언어의 어휘 토큰을 명세할 수 있다(Figure 1.2). 각 토큰에 대해서는 어떤 토큰 타입이 인식되었는지를 보고하는 코드[2]를 제공한다.

다섯 번째 줄의 규칙은 주석이나 공백을 인식하지만, 파서에 전달하지 않는다. 대신 공백은 버려지고 렉서가 다시 진행된다(`continue`가 하는 일이다). 이 렉서의 주석은 두 개의 대시(`--`)로 시작하며, 알파벳 문자만 포함하고, 개행 문자에서 끝난다.

마지막으로, 어휘 명세는 항상 완전해야 하며, 입력의 어떤 접두부도 반드시 매칭해야 한다. 이를 위해 "임의의 한 문자"를 매칭하는 규칙을 추가할 수 있으며, 이 경우 "잘못된 문자(illegal character)" 오류 메시지를 출력한 후 렉서를 계속 진행하게 한다.

이러한 규칙들은 다소 모호할 수 있다. 예를 들어, `if8`은 하나의 식별자로 인식되는가, 아니면 `if`와 `8`이라는 두 개의 토큰으로 인식되는가? 문자열 `if 89`는 식별자로 시작하는가, 아니면 예약어로 시작하는가? Lex와 같은 어휘 분석기 생성기들은 이러한 모호성을 해소하기 위해 두 가지 중요한 규칙을 사용한다.

1. **최장 매치(Longest match):** 입력에서 가능한 한 가장 긴 접두부를 다음 토큰으로 선택한다.
2. **규칙 우선순위(Rule priority):** 동일한 최장 접두부에 대해 가장 먼저 매칭되는 정규 표현식이 그 토큰 타입을 결정한다. 따라서 정규 표현식 규칙을 작성하는 순서가 중요하다.

따라서 `if8`은 최장 매치 규칙에 의해 식별자로 인식되고, `if`는 규칙 우선순위에 의해 예약어로 인식된다.

1.3 유한 오토마타(Finite Automata)

정규 표현식은 어휘 토큰을 명세하는 데 편리하지만, 이를 실제 컴퓨터 프로그램으로 구현할 수 있는 알고리즘(formalism)이 필요하다. 이를 위해 **유한 오토마타(finite automata)**를 사용할 수 있다. (참고: *automata*의 단수형은 *automaton*이다.)

유한 오토마타는 유한한 상태 집합을 가진다. 상태들 사이에는 간선(edge)이 존재하며, 각 간선은 기호(symbol)로 라벨링된다. 하나의 상태는 시작 상태(start state)로 지정되고, 일부 상태들은 종료 상태(final state)로 구별된다.

[2]현재 코드는 ML 언어로 작성되었음

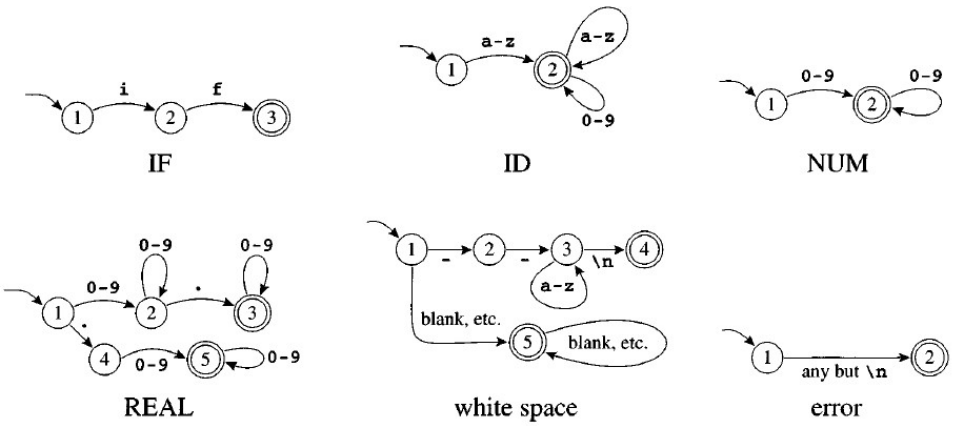


Figure 1.3: 어휘 토큰을 위한 유한 오토마타 예시
상태(state)는 원으로 표시되며, 종료 상태(final state)는 이중 원(double circle)으로 표시된다. 시작 상태는 어디서도 들어오는 것이 없는 화살표로 표시된다. 여러 문자로 라벨링된 간선은 여러 개의 병렬 간선을 줄여 쓴 표기이다.

Figure 1.3는 몇 가지 유한 오토마타를 보여준다. 상태들은 논의를 편리하게 하기 위해 번호가 붙어 있으며, 각 경우 시작 상태는 1번이다. 여러 문자로 라벨링된 간선은 여러 개의 병렬 간선을 줄여 쓴 것이다. 예를 들어 ID 인식기에서 실제로는 상태 1에서 2로 가는 26개의 간선이 있으며, 각각은 다른 알파벳 문자로 라벨링되어 있다.

결정적 유한 오토마타(DFA)에서는 동일한 상태에서 나가는 간선들이 동일한 기호를 공유할 수 없다. DFA는 문자열을 다음과 같이 수용(accept)하거나 거부(reject)한다. 시작 상태에서 입력 문자열의 각 문자를 읽을 때마다 해당 문자가 라벨링된 정확히 하나의 간선을 따라 다음 상태로 이동한다. n개의 문자를 가진 문자열에 대해 n번의 전이를 수행한 뒤 종료 상태에 도달하면 문자열은 수용된다. 종료 상태가 아니거나, 도중에 해당 문자가 라벨링된 간선이 없으면 문자열은 거부된다. 오토마타가 인식하는 언어는 오토마타가 수용하는 문자열들의 집합이다.

예를 들어, ID 오토마타가 인식하는 언어의 모든 문자열은 반드시 문자(letter)로 시작해야 한다. 임의의 단일 문자는 상태 2로 이어지며, 상태 2는 종료 상태이므로 단일 문자 문자열은 수용된다. 상태 2에서 어떤 문자나 숫자를 읽어도 다시 상태 2로 돌아오기 때문에, 문자 뒤에 임의의 개수의 문자와 숫자가 이어진 문자열도 수용된다.

실제로 Figure 1.3에 나온 기계들은 Figure 1.2의 정규 표현식과 동일한 언어를 수용한다.

이들은 여섯 개의 독립적인 오토마타인데, 어떻게 하나의 기계로 결합하여 어휘 분석기로 사용할 수 있을까? 다음 절에서는 이를 형식적으로 다루겠지만, 여기서는 임시적으로(ad hoc) 설명한다. Figure 1.4은 그러한 결합된 기계를 보여준다. 각 종료 상태에는 해당 토큰 타입이 라벨링되어야 한다.

이 기계의 상태 2는 IF 기계의 상태 2와 ID 기계의 상태 2의 성질을 모두 가진다. 후자가 종료 상태이므로, 결합된 상태도 종료 상태여야 한다. 상태 3은 IF 기계의 상태 3과 ID 기계의 상태 2에 해당하며, 둘 다 종료 상태이므로 규칙 우선순위(rule priority)에 따라 IF로 라벨링한다. 즉, 이 토큰은 식별자가 아니라 예약어로 인식되도록 한다.

이 기계는 전이 행렬(transition matrix)로 인코딩할 수 있다. 전이 행렬은 상태 번호와 입력 문자로 색인되는 2차원 배열(벡터의 벡터)이다. 이때 모든 문자에 대해 자기 자신으로 이동하는 "죽은 상태(dead

state)"(상태 0)가 필요하다. 이 상태는 간선이 존재하지 않는 경우를 표현하는 데 사용된다.

```
val edges =
 vector[          (*      ···0 1 2···-···e f g h i j···*)
(* state 0 *) vector[0,0,···0,0,0···0···0,0,0,0,0,0···],
(* state 1 *) vector[0,0,···7,7,7···9···4,4,4,4,2,4···],
(* state 2 *) vector[0,0,···4,4,4···0···4,3,4,4,4,4···],
(* state 3 *) vector[0,0,···4,4,4···0···4,4,4,4,4,4···],
(* state 4 *) vector[0,0,···4,4,4···0···4,4,4,4,4,4···],
(* state 5 *) vector[0,0,···6,6,6···0···0,0,0,0,0,0···],
(* state 6 *) vector[0,0,···6,6,6···0···0,0,0,0,0,0···],
(* state 7 *) vector[0,0,···7,7,7···0···0,0,0,0,0,0···],
   et cetera
        ]
```

또한 상태 번호를 동작(action)에 매핑하는 "종결성(finality) 배열"도 필요하다. 예를 들어, 종료 상태 2는 동작 ID에 매핑된다.

최장 매치 인식 (Recognizing the Longest Match)

이 테이블을 사용하여 문자열을 수용할지 거부할지를 결정하는 것은 간단하지만, 어휘 분석기의 임무는 입력의 접두부 중 유효한 토큰이 되는 가장 긴 문자열(longest match)을 찾는 것이다. 전이를 해석하면서 렉서는 지금까지 발견된 최장 매치와 그 위치를 추적해야 한다.

최장 매치를 추적한다는 것은, 마지막으로 종료 상태에 있었던 시점과 위치를 기억하는 것을 의미한다. 이를 위해 두 변수를 사용한다: `Last-Final` (최근에 도달한 종료 상태의 번호)과 `Input-Position-at-Last-Final` (그 시점의 입력 위치)이다.

종료 상태에 도달할 때마다 렉서는 이 변수를 갱신한다. 죽은 상태(출력 전이가 없는 비종결 상태)에 도달하면, 이 변수들을 이용해 어떤 토큰이 매칭되었고 어디에서 끝났는지를 알 수 있다.

Figure 1.5는 최장 매치를 인식하는 어휘 분석기의 동작을 보여준다. 주목할 점은 현재 입력 위치가 최근 종료 상태에 도달했던 위치보다 훨씬 뒤일 수도 있다는 것이다.

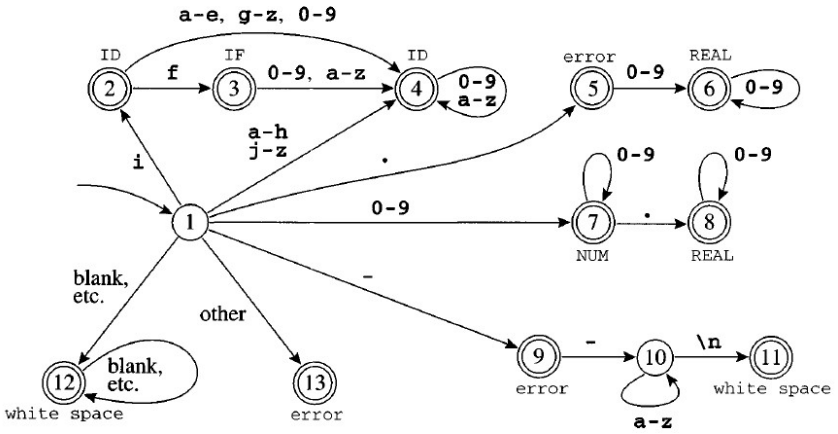


Figure 1.4: 여러 오토마타를 결합한 어휘 분석기 예시

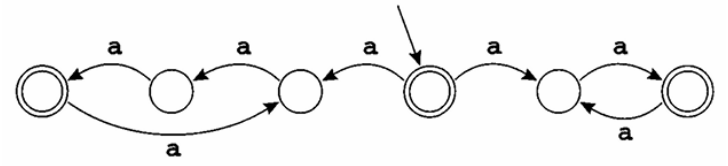
Last Final	Current State	Current Input	Accept Action
0	1	⊤⊥if --not-a-com	
2	2	|⊤⊥if --not-a-com	
3	3	|if⊤⊥ --not-a-com	
3	0	|if⊤ ⊥--not-a-com	*return* IF
0	1	if⊤⊥ --not-a-com	
12	12	if |⊤⊥--not-a-com	
12	0	if |⊤ ⊥not-a-com	*found white space; resume*
0	1	if ⊤⊥--not-a-com	
9	9	if |⊤⊥not-a-com	
9	10	if |⊤n⊥ot-a-com	
9	10	if |⊤no⊥t-a-com	
9	10	if |⊤not⊥-a-com	
9	0	if |⊤not ⊥a-com	*error, illegal token '-'; resume*
0	1	if ⊤⊥not-a-com	
9	9	if |⊤⊥not-a-com	
9	0	if |⊤ ⊥not-a-com	*error, illegal token '-'; resume*

Figure 1.5: 최장 매치를 인식하는 어휘 분석기의 동작: Figure 1.4의 오토마타는 여러 토큰을 인식한다. 기호 |는 어휘 분석기가 연속적으로 호출될 때의 입력 위치를 나타내며, 기호 ⊥은 오토마타의 현재 위치를 나타내고, ⊤은 인식기가 마지막으로 종료 상태에 있었던 위치를 나타낸다.

1.4 비결정적 유한 오토마타

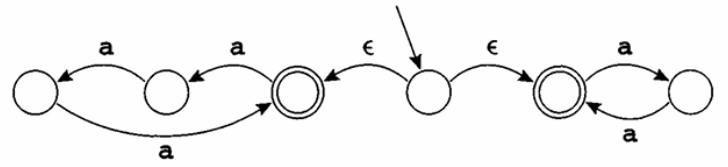
비결정적 유한 오토마타(NFA, nondeterministic finite automaton)는 동일한 기호로 라벨링된 여러 개의 간선 중 하나를 선택하여 따라갈 수 있는 오토마타를 말한다. 또한 입력에서 문자를 소비하지 않고도 따라갈 수 있는 특별한 ϵ-간선이 있을 수도 있다.

다음의 비결정적 유한 오토마타 예시를 보자.



시작 상태에서 입력 문자 a를 읽을 때, 오토마타는 왼쪽이나 오른쪽으로 이동할 수 있다. 왼쪽을 선택하면, 길이가 3의 배수인 문자열이 받아들여진다. 오른쪽을 선택하면, 길이가 짝수인 문자열이 받아들여진다. 따라서 이 NFA가 인식하는 언어는 a들의 길이가 2의 배수이거나 3의 배수인 문자열 집합이다. 첫 전이에서 이 기계는 어느 쪽으로 갈지를 선택해야 한다. 어떤 경로를 택하든 수용할 수 있는 경우가 존재한다면 반드시 문자열을 받아들여야 하므로, 불가피하게 "추측"을 해야 하며, 항상 올바르게 추측해야 한다.

ϵ-간선은 입력 기호를 소모하지 않고도 따라갈 수 있다. 다음은 동일한 언어를 받아들이는 다른 비결정적 유한 오토마타이다.

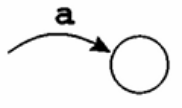


시작 상태에서 어느 ϵ 간선을 택할지 선택해야 한다. 만약 어떤 상태에 ϵ-간선과 기호로 라벨링된 간선이 함께 있다면, 기계는 입력 기호를 소비해 해당 간선을 따라가거나, ϵ-간선을 따라갈 수 있다.

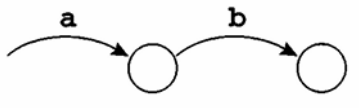
정규 표현식을 NFA로 변환하기

비결정적 오토마타(NFA)는 유용한 개념이다. 왜냐하면 (정적이고 선언적인) 정규 표현식을 (시뮬레이션 가능한, 준-실행 가능한) NFA로 쉽게 변환할 수 있기 때문이다. 변환 알고리즘은 각 정규 표현식을 하나의 꼬리(tail, 시작 간선)와 머리(head, 종료 상태)를 가진 NFA로 바꾼다.

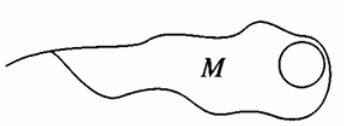
예를 들어 단일 기호 정규 표현식 a는 다음과 같은 NFA로 변환된다.



정규 표현식 ab는 a와 b의 NFA를 연결하여 구성된다. a의 머리를 b의 꼬리에 연결하면, 결과적으로 a에서 b로 이어지는 전이가 생긴다.



어떤 정규 표현식 M도 꼬리와 머리를 가진 NFA로 바꿀 수 있다[3]. 이 꼬리–머리 구조를 이용해 더 큰 정규 표현식도 단계적으로 NFA로 합성할 수 있다.



이 변환은 귀납적으로 정의할 수 있다. 정규 표현식이 원시적(단일 기호나 ϵ)일 수도 있고, 더 작은 표현식들을 결합해 만들어질 수도 있다. 마찬가지로 NFA도 원시적일 수 있고, 작은 NFA들을 조합해 구성될 수도 있다.

Figure 1.6는 정규 표현식을 비결정적 오토마타로 변환하는 규칙을 보여준다. Figure 1.2의 일부 토큰들(IF, ID, NUM, error)의 정규식들에 대해 변환을 적용하면, 각 표현식이 NFA로 바뀌며, 각 NFA의 머리 상태는 서로 다른 토큰 타입으로 종료 상태로 표시된다. 모든 꼬리는 새로운 시작 노드에 연결된다. 동등한 NFA 상태들을 병합한 결과는 Figure 1.7과 같다.

[3]이 형태의 그림을 그림 1.6의 정규 표현식을 NFA로 변환하는 방법을 설명할 때 기본 요소로 취급한다. M은 정규 표현식이고, 그 안쪽의 작은 원은 종료 상태(final state), 왼쪽 곡선은 꼬리(tail, 시작 지점)이다. 왼쪽 곡선의 꼬리(tail, 시작 지점)에서 M을 거쳐 머리(head, 종료 지점)로 이어지는 흐름을 나타냅니다.

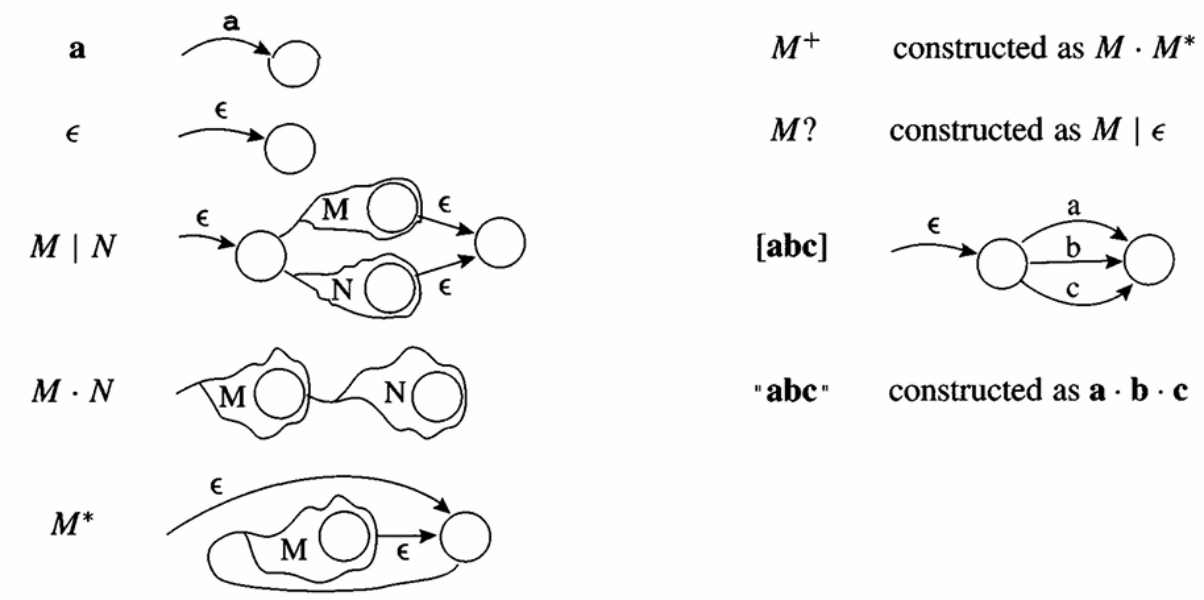


Figure 1.6: 정규 표현식을 NFA로 변환

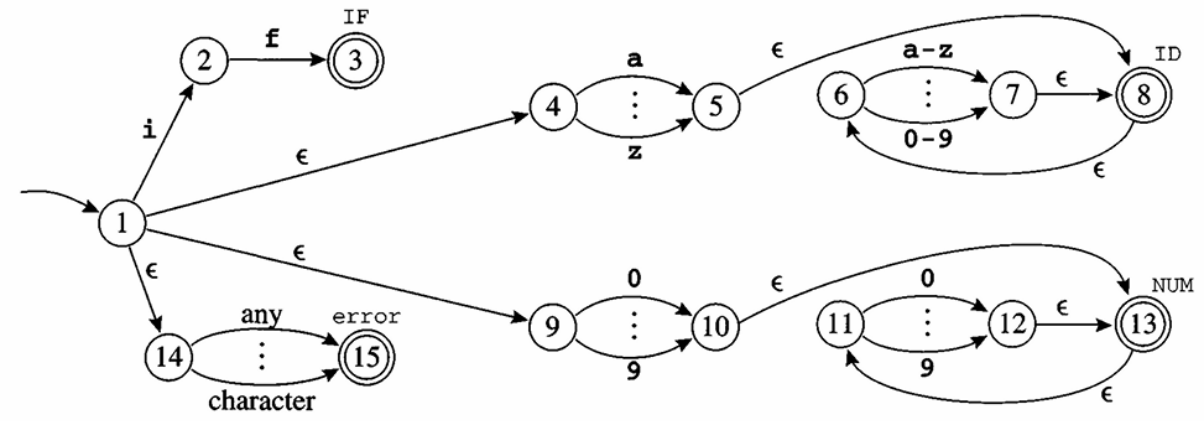


Figure 1.7: 토큰 IF, ID, NUM, error를 변환한 NFA의 결합 결과

NFA를 DFA로 변환하기

1.3절에서 보았듯, DFA는 프로그램으로 구현하기 쉽다. 하지만 NFA는 "추측" 기능이 필요해 구현이 어렵다. 이를 피하기 위해, 가능한 모든 경로를 동시에 따라가는 방식으로 시뮬레이션할 수 있다.

모든 가능성을 한꺼번에 시도함으로써 추측할 필요를 피할 수 있다. 예를 들어 Figure 1.7의 NFA를 문자열 `"in"`에 대해 시뮬레이션한다고 하자. 시작 상태는 1이다. 이때 ϵ-간선 중에서 어떤 것을 따라갈지 추측하기보다, ϵ-전이를 따라갈 수 있는 모든 상태를 고려하면, 가능한 상태 집합은 $\{1, 4, 9, 14\}$이다. 이를 ϵ-closure라고 한다. ϵ-closure를 계산하면, 입력을 소비하지 않고 도달 가능한 모든 상태를 이미 찾은 것이다. 입력을 실제로 읽기 전에는 이 외의 상태로 절대 갈 수 없다.

다음으로 문자 i를 읽는다. 상태 1에서 2로, 4에서 5로, 9에서는 이동 불가, 14에서 15로 이동한다. 따라서 가능한 상태 집합은 $\{2, 5, 15\}$이다. 여기에 다시 ϵ-closure를 적용하면 $\{2, 5, 6, 8, 15\}$가 된다.

문자 `n`에서, 우리는 상태 6에서 7로, 상태 2에서는 아무 곳에도, 상태 5에서도 아무 곳에도, 상태 8에서

도 아무 곳에도, 상태 15에서도 아무 곳에도 도달하지 못한다. 따라서 가능한 상태 집합은 {7}이 된다. 그 ϵ-closure는 $\{6, 7, 8\}$이다.

이제 우리는 문자열 `in`의 끝에 도달하였다. NFA는 종료 상태에 있는가? 앞서 구한 ϵ-clousre에 포함된 8이 종료 상태이다. 따라서 `in`은 ID 토큰이다.

ϵ-clousre를 다음과 같이 수학으로 정리할 수 있다.

- **edge**(s, c)는 상태 s에서 기호 c가 붙은 하나의 간선을 거쳐 갈 수 있는 모든 상태 집합이다.
- **closure**(S)는 상태 집합 S에서 ϵ-간선만 따라가 도달할 수 있는 상태들의 집합이다.

수학적으로, ϵ-간선을 따라가는 개념은 다음과 같이 표현할 수 있다. 즉, **closure**(S)는 집합 T 중에서 다음 조건을 만족하는 가장 작은 집합이다.

$$T = S \cup \left(\bigcup_{s \in T} \mathbf{edge}(s, \epsilon) \right).$$

반복 과정으로 T를 구할 수 있다.

$T \leftarrow S$;
repeat
  $T' \leftarrow T$;
  $T \leftarrow T' \cup \left(\bigcup_{s \in T'} \mathbf{edge}(s, \epsilon) \right)$;
until $T = T'$;

왜 이 알고리즘이 동작하는가? T는 각 반복에서 오직 커질 수만 있으므로, 최종적으로 얻어지는 T는 반드시 S를 포함해야 한다. 만약 어떤 반복 단계에서 $T = T'$라면, T는 또한 $\bigcup_{s \in T'} \mathbf{edge}(s, \epsilon)$도 포함해야 한다. 마지막으로, NFA에는 유한 개의 서로 다른 상태만 존재하므로 이 알고리즘은 반드시 종료한다.

이제 위에서 설명한 방식으로 NFA를 시뮬레이션한다고 하자. 만약 우리가 $d = \{s_t, s_k, s_i\}$라는 NFA 상태 집합에 있다고 하자. d에서 시작하여 입력 기호 c를 소비하면, 우리는 새로운 NFA 상태 집합에 도달하게 된다. 우리는 이 집합을 **DFAedge**(d, c)라고 부른다.

$$\mathbf{DFAedge}(d, c) \;=\; \mathbf{closure}\left(\bigcup_{s \in d} \mathbf{edge}(s, c) \right)$$

DFAedge를 사용하면, NFA 시뮬레이션 알고리즘을 더 엄밀하게 쓸 수 있다. 만약 NFA의 시작 상태가 s_1이고 입력 문자열이 $c_1, \ldots, c_k$라면, 알고리즘은 다음과 같다.

$d \leftarrow closure(\{s_1\})$;
for $i \leftarrow 1$ **to** k **do**
  $d \leftarrow DFAedge(d, c_i)$;
end

DFAedge로 상태 집합들을 구하기는 비용이 많이 들기 때문에, 어휘 분석 중인 소스 프로그램의 각 문자마다 이를 수행하기에는 지나치게 비효율적이다. 그러나 모든 상태 집합 계산을 사전에 해둘 수 있다. NFA로부터 DFA를 구성하는데, 각 NFA 상태 집합이 하나의 DFA 상태에 대응하도록 한다. NFA가 유한한 수 n개의 상태를 가진다면, DFA 역시 유한한 수의 상태를 가지며, 그 수는 최대 2^n개이다.

closure와 **DFAedge** 알고리즘이 주어지면 DFA 구성이 쉬워진다. DFA의 시작 상태 d_1은 NFA 시뮬레이션 알고리즘에서와 같이 **closure**(s_1)이다. 추상적으로, 어떤 문자 c에 대해 $d_j = \textbf{DFAedge}(d_i, c)$일 때 d_i에서 d_j로의 간선이 존재한다.

NFA로부터 DFA를 만드는 알고리즘은 다음과 같다. Σ는 입력 알파벳이다.

$states[0] \leftarrow \{\}$;
$states[1] \leftarrow$ **closure**$(\{s_1\})$;
$p \leftarrow 1$;
$j \leftarrow 0$;
while $j \leq p$ **do**
  foreach $c \in \Sigma$ **do**
    $e \leftarrow$ **DFAedge**$(states[j], c)$;
    if $e = states[i]$ *for some* $i \leq p$ **then**
      $trans[j, c] \leftarrow i$;
    end
    else
      $p \leftarrow p + 1$;
      $states[p] \leftarrow e$;
      $trans[j, c] \leftarrow p$;
    end
  end
  $j \leftarrow j + 1$;
end

이 알고리즘은 DFA의 도달 불가능한 상태들을 방문하지 않는다. 이는 매우 중요한데, 원칙적으로 DFA는 2^n개의 상태를 가질 수 있지만, 실제로는 시작 상태로부터 도달 가능한 상태가 대략 n개 정도에 불과하기 때문이다. 따라서 컴파일러나 구문 분석 도구의 일부가 될 DFA 해석기의 전이 테이블 크기가 지수에 비례해서 커지지 않도록 하는 것이 중요하다.

이때 DFA에서 어떤 상태 d는 $states[d]$에 속한 NFA 상태들 중 하나라도 종료 상태라면 종료 상태로 지정한다. 어떤 상태를 종료 상태로 표시하는 것만으로는 부족하다. 그 종료 상태에서 어떤 토큰이 인식되는지도 명시해야 한다. 경우에 따라서는 $states[d]$에 속한 여러 NFA 상태들이 종료 상태일 수도 있다. 이 경우, 우리는 어휘 명세를 이루는 정규 표현식들의 목록에서 가장 먼저 나타난 토큰 유형으로 d를 라벨링한다. 이것이 바로 규칙 우선순위(rule priority)를 구현하는 방법이다.

DFA가 구성된 이후에는 `states` 배열은 버려도 되며, `trans` 배열이 어휘 분석에 사용된다. 그림 1.7의 NFA에 DFA 구성 알고리즘을 적용하면 그림 1.8의 오토마타가 얻어진다.

이 오토마타는 최적이 아니다. 즉, 같은 언어를 인식하는 더 작은 오토마타가 존재한다. 일반적으로, 두 상태 s_1과 s_2에 대해, 기계가 s_1에서 시작하여 문자열 σ를 받아들일 때 그리고 오직 그럴 때에만 s_2에서 시작했을 때도 σ를 받아들이면, 우리는 s_1과 s_2가 동치라고 말한다. 이는 Figure 2.8의 상태 $\{5, 6, 8, 15\}$와 $\{6, 7, 8\}$에 대해 분명히 성립한다. 또한 상태 $\{10, 11, 13, 15\}$와 $\{11, 12, 13\}$에 대해서도 성립한다. 두 동치

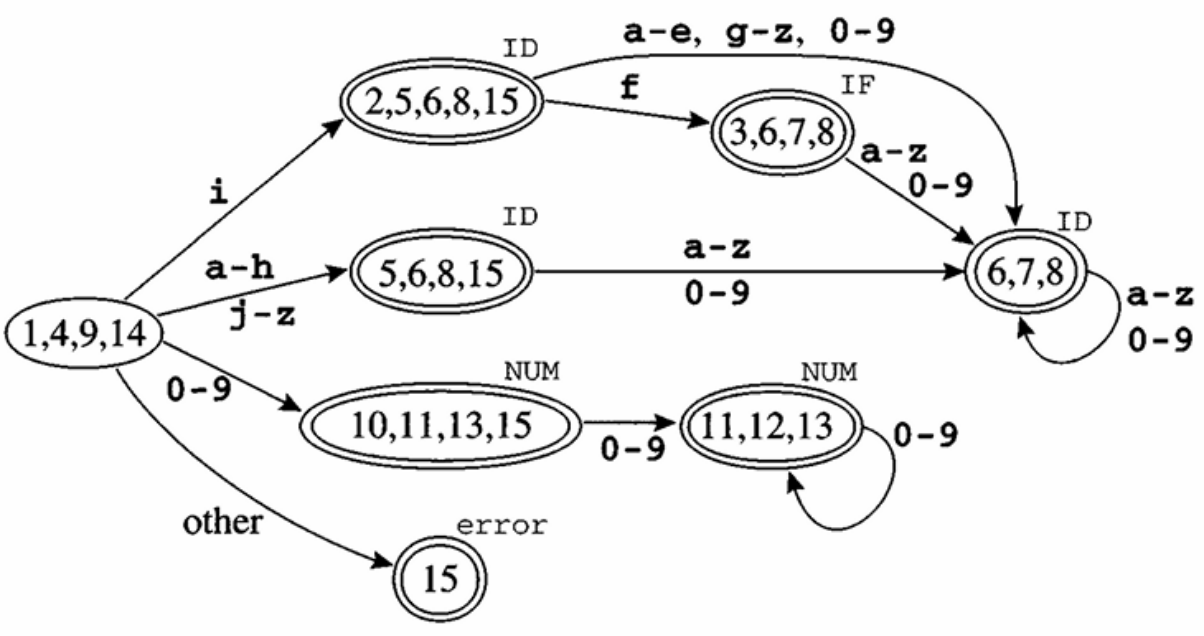
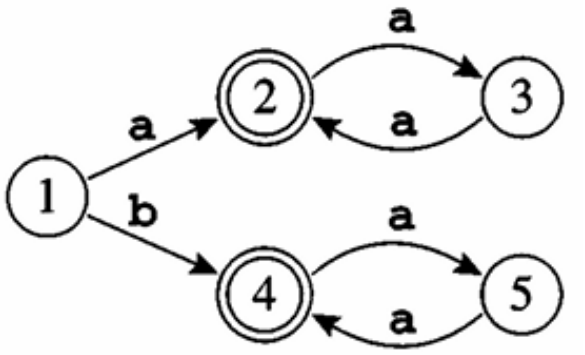


Figure 1.8: NFA를 DFA로 변환

상태 s_1과 s_2가 있는 오토마타에서는, s_2로 들어오는 모든 간선을 s_1로 향하게 바꾸고 s_2를 삭제할 수 있다.

그렇다면 동치 상태를 어떻게 찾을 수 있을까? 분명히, s_1과 s_2가 모두 종료 상태이거나 모두 비종료 상태이고, 임의의 기호 c에 대해 $trans[s_1, c] = trans[s_2, c]$라면, 이 두 상태는 동치이다. $\{10, 11, 13, 15\}$와 $\{11, 12, 13\}$은 이 조건을 만족한다. 동일한 입력에 대해 똑같은 다음 상태로 가야 한다는 이 조건만으로는 충분히 일반적이지 않다. 다음 오토마타를 예로 살펴 보면, 상태 2와 상태 4가 동치이지만, $trans[2, a] = 3 \neq 5 = trans[4, a]$이기 때문이다.



핵심은 두 상태가 동치라는 것은, 그 상태에서 시작했을 때 받아들이는 언어가 완전히 동일하다라는 점이다. 상태 3과 상태 5는 둘 다 비종료 상태이고, 또 a를 받으면 다시 각각 2와 4로 돌아간다. 따라서 2에서 시작했을 때 인식하는 문자열 집합과 4에서 시작했을 때 인식하는 문자열 집합은 똑같다. 즉, 겉보기 전이는 다르지만, 결과적으로는 같은 언어를 인식하기 때문에 상태 2와 4는 동치이다.

DFA를 구성한 뒤에는, 동치 상태를 찾아내어 최소화(minimization) 알고리즘[4]을 적용하는 것이 유용하다.

[4] https://en.wikipedia.org/wiki/DFA_minimization

1.5 Further Reading

Lex는 정규 표현식을 기반으로 한 최초의 어휘 분석기 생성기였다 [Les75]; 오늘날까지도 널리 사용되고 있다. ϵ-closure 계산은 아직 ϵ-전이를 검사하지 않은 상태들을 큐나 스택에 유지함으로써 더 효율적으로 수행할 수 있다 [ASU86]. 정규 표현식은 NFA를 거치지 않고 직접 DFA로 변환될 수도 있다 [MY60, ASU86].

DFA 전이 표는 매우 크고 희소(sparse)할 수 있다. 이를 단순한 2차원 행렬 {states × symbols}로 표현하면 지나치게 많은 메모리를 차지한다. 실제로는 표를 압축하여 메모리 사용량을 줄이는데, 이는 필요한 메모리를 감소시키지만 다음 상태를 찾는 데 걸리는 시간을 증가시킨다 [ASU86].

어휘 분석기는 자동 생성되었든 수작업으로 작성되었든 간에 입력을 효율적으로 관리해야 한다. 물론 입력은 버퍼링되며, 한 번에 많은 문자를 읽어온 뒤 버퍼 안에서 문자를 하나씩 처리한다. 어휘 분석기는 각 문자마다 버퍼의 끝에 도달했는지 확인해야 한다. 버퍼 끝에 어떤 토큰에도 속할 수 없는 특수 문자(sentinel)를 두면, 어휘 분석기는 문자마다가 아니라 토큰마다 한 번만 버퍼 끝을 검사하면 된다 [ASU86]. Gray [Gra88]는 토큰마다가 아니라 한 줄(line)마다 단 한 번만 검사하는 방식을 사용했지만, 이 방식은 개행 문자를 포함하는 토큰을 처리할 수 없었다. Bumbulis와 Cowan [BC93]은 DFA의 각 사이클마다 한 번만 검사를 수행하여, DFA 경로가 긴 경우 문자마다 검사를 수행하는 것보다 검사 횟수를 줄였다.

자동으로 생성된 어휘 분석기는 종종 느리다는 비판을 받는다. 원칙적으로 유한 오토마타의 동작은 매우 단순하므로 효율적이어야 하지만, 전이 표를 해석하는 과정에서 오버헤드가 발생한다. Gray [Gra88]는 DFA를 직접 실행 코드로 변환하여 (상태를 case 문으로 구현) 수작업으로 작성한 어휘 분석기만큼 빠르게 동작할 수 있음을 보였다. Flex, 즉 "빠른 어휘 분석기 생성기" [Pax95]는 Lex보다 훨씬 빠르다.

1.6 연습문제

다음과 같은 렉시컬 명세의 분석기를 작성하시오. 이 분석기를 DFA로 작성하시고, 각 상태에 어느 액션이 실행될지 표시하시오. (참고: 그림 1.8. 단 상태에 토큰이 아닌 액션을 표시)

$$
\begin{array}{ll}
(aba)^{+} & \text{(action 1);} \\
(a(b^{*})a) & \text{(action 2);} \\
(a|b) & \text{(action 3);}
\end{array}
$$

작성한 렉서로 문자열 `abaabbaba`을 분석하시오. 각 세부 분석 단계를 작성하시오.

Chapter 2

구문 분석 (Parsing): 어휘 간 구조 분석-파싱

Andrew W. Appel, Modern Compiler Implementation in ML, Cambridge University Press (Chapter 3∼5)

> syn-tax: 단어들이 구, 절, 문장을 이루도록 배열되는 방식.
>
> *웹스터 사전(Webster's Dictionary)*

아래는 정규 표현식을 대신하는 약어(abbreviation)를 선언하는 방법이다. 이는 상당히 편리하고 여러모로 흥미롭게 사용할 수 있다.

```
digits = [0-9]+
sum    = (digits "+")* digits
```

이 정규 표현식들은 $28 + 301 + 9$와 같은 합을 정의한다. 그러나 이제 다음을 고려해 보자:

```
digits = [0-9]+
sum    = expr"+"+expr
expr   = "(" sum ")" | digits
```

이는 다음과 같은 꼴의 식들을 정의하려는 것이다.

```
(109+23)
61
(1+(250+3))
```

여기서 모든 괄호는 짝이 맞아야 한다. 하지만 유한 오토마톤으로는 짝이 맞는 괄호를 인식할 수 없다. 왜냐하면 N개의 상태를 가진 기계는 괄호의 중첩 깊이가 N보다 큰 것을 기억할 수 없기 때문이다. 따라서 `sum`과 `expr`는 정규 표현식이 될 수 없다.

앞서 digits와 같은 정규 표현식 약어를 구현할 수 있었던 이유는 정규 표현식에서 digits가 등장하는 모든 곳에 오른쪽 항 ($[0-9]+$)을 그대로 대체해서 진행하면 되기 때문이다.

sum-expr 언어에서는 이런 방식이 통하지 않는다. 먼저 expr에 sum을 대체하자.

```
expr = "(" expr "+" expr ")" | digits
```

그 다음 expr을 자기 자신 안에 대체하려 하면 다음과 같이 된다.

```
expr = "(" ("(" expr "+" expr ")" | digits) "+" expr ")" | digits
```

이제 오른쪽 항에는 expr가 이전만큼, 아니 그보다 더 많이 등장한다!

따라서 약어라는 개념은 정규 표현식 언어의 표현력을 증가시키지는 않는다. 즉, 약어가 재귀적(혹은 sum과 expr처럼 상호 재귀적)일 경우 외에는 새로운 언어가 생기는 것은 아니다.

재귀를 통해 얻는 추가 표현력이 바로 우리가 구문 분석(Parsing)이 필요한 이유이다. 또한 일단 약어에서 재귀를 사용할 수 있다면, 최상위 수준의 식을 정의할 때만 필요하고 그 외에는 선택(alternation)이 필요 없다. 예를 들어

```
expr = ab(c | d)e
```

는 보조 정의를 도입해 다음과 같이 쓸 수 있다:

```
aux = c | d
expr = a b aux e
```

사실, 같은 기호에 대해 여러 번 정의할 수 있게 하면 아예 선택 기호를 쓰지 않아도 된다.

```
aux = c
aux = d
expr = a b aux e
```

클린 내포(Kleene closure)도 꼭 필요하지 않다. 예를 들어

```
expr = (a b c)*
```

는 다음과 같이 바꿀 수 있다:

```
expr = (a b c) expr
expr = \epsilon
```

결국 남는 것은 매우 단순한 표기법인데, 이것을 *문맥 자유 문법(context-free grammar)*이라고 한다. 정규 표현식이 정적이고 선언적인 방식으로 어휘 구조를 정의하는 데 사용될 수 있듯, 문법은 구문 구조를 선언적으로 정의한다. 하지만 문법으로 기술된 언어를 분석하려면 유한 오토마톤보다 강력한 것이 필요하다. 사실 문법은 어휘 토큰의 구조를 기술하는 데도 쓸 수 있지만, 그 목적에는 정규 표현식이 더 간결하고 충분하다.

1 $S \to S; S$
2 $S \to id := E$
3 $S \to print(L)$
4 $E \to id$
5 $E \to num$
6 $E \to E + E$
7 $E \to (S, E)$
8 $L \to E$
9 $L \to L, E$

Figure 2.1: 직선형(straight-line) 프로그램의 구문 문법.

2.1 문맥 자유 문법(Context-Free Grammars)

언어는 문자열들의 집합이다. 각 문자열은 유한 알파벳으로부터 뽑은 유한 개의 기호들의 열이다. 구문 분석에서 문자열은 소스 프로그램이고, 기호는 어휘 토큰(lexical tokens)이며, 알파벳은 어휘 분석기가 반환하는 토큰 종류들(token types)의 집합이다.

문맥 자유 문법으로 언어를 기술한다. 문법은 다음과 같은 생성 규칙(production) 집합이다.

```
symbol -> symbol symbol ... symbol
```

오른쪽에는 0개 이상의 기호가 온다. 각 기호는 *터미널*이거나, *비터미널*이다. 터미널은 언어의 문자열 알파벳에 속하는 토큰이고, 비터미널은 어떤 생성 규칙의 왼쪽에 위치하는 기호다. 토큰은 생성 규칙의 왼쪽에 올 수 없다. 마지막으로, 어떤 비터미널 하나가 문법의 *시작 기호*로 지정된다.

그림 2.1의 문법은 직선형 프로그램의 문법 예시다. 시작 기호는 S이다. 시작 기호가 명시되지 않을 경우 관례상 첫 번째 생성 규칙의 왼쪽 비터미널을 시작 기호로 한다. 터미널 기호는 `id`, `print`, `num`, `,`, `+`, `(`, `)`, `;`, `:=`이고, 비터미널은 `S`, `E`, `L`이다. 이 문법의 언어에 속하는 문장 예시이다.

```
id := num; id := id(id:= num + num, id)
```

어휘 분석을 하기 전의 소스 텍스트는

```
a := 7; bc := d(d := 5 + 6, d)
```

였다. 토큰 종류(터미널 기호)는 `id`, `num`, `:=` 등이 되고, 이름(`a`, `b`, `c`, `d`)와 숫자(`7`, `5`, `6`)은 일부 토큰(`id`, `num`)에 연관된 의미 값이다.

유도(derivation)

이 문장이 문법의 언어에 속한다는 것을 보이려면 *유도*를 수행하면 된다. 시작 기호에서 출발해, 비터미널을 반복적으로 하나씩 그 오른쪽 항으로 치환하는 것이다. 이 과정은 그림 2.2의 유도 예시와 같다.

같은 문장에 대해 여러 다른 유도가 있을 수 있다. *좌측 유도*는 항상 가장 왼쪽의 비터미널을 확장하는 경우이고, *우측 유도*는 항상 가장 오른쪽 비터미널을 확장하는 경우다.

그림 2.2의 예시는 좌측 유도도 아니고 우측 유도도 아니다. 이 문장을 좌측 유도하면 다음과 같이 전개될 것이다.

$\underline{S}$

$S\ ;\ \underline{S}$

$\underline{S}\ ;\ \texttt{id} := E$

$\texttt{id} := \underline{E}\ ;\ \texttt{id} := E$

$\texttt{id} := \texttt{num}\ ;\ \texttt{id} := \underline{E}$

$\texttt{id} := \texttt{num}\ ;\ \texttt{id} := E + \underline{E}$

$\texttt{id} := \texttt{num}\ ;\ \texttt{id} := \underline{E} + (S, E)$

$\texttt{id} := \texttt{num}\ ;\ \texttt{id} := \texttt{id} + (\underline{S}, E)$

$\texttt{id} := \texttt{num}\ ;\ \texttt{id} := \texttt{id} + (\texttt{id} := \underline{E}, E)$

$\texttt{id} := \texttt{num}\ ;\ \texttt{id} := \texttt{id} + (\texttt{id} := E + E, \underline{E})$

$\texttt{id} := \texttt{num}\ ;\ \texttt{id} := \texttt{id} + (\texttt{id} := \underline{E} + E, id)$

$\texttt{id} := \texttt{num}\ ;\ \texttt{id} := \texttt{id} + (\texttt{id} := \texttt{num} + \underline{E}, \texttt{id})$

$\texttt{id} := \texttt{num}\ ;\ \texttt{id} := \texttt{id} + (\texttt{id} := \texttt{num} + \texttt{num}, \texttt{id})$

Figure 2.2: 유도 예시

$\underline{S}$

$\underline{S}\ ;\ S$

$\texttt{id} := \underline{E}\ ;\ S$

$\texttt{id} := \texttt{num}\ ;\ \underline{S}$

$\texttt{id} := \texttt{num}\ ;\ \texttt{id} := \underline{E}$

$\texttt{id} := \texttt{num}\ ;\ \texttt{id} := \underline{E} + E$

$\vdots$

구문 트리(parse tree)

*구문 트리*는 유도 과정에서 각 기호를 그것이 유래한 기호와 연결하여 만든다. 그림 2.3에 예시가 있다. 서로 다른 두 유도가 같은 구문 트리를 가질 수 있다.

모호한 문법(Ambiguous Grammars)

한 문법이 하나의 문장을 두 가지 다른 구문 트리로 유도할 수 있다면, 그 문법은 *모호하다*고 한다. 문법 2.1은 모호하다. 예컨대 문장 $id := id + id + id$에는 두 가지 구문 트리가 존재한다(그림 2.4).

문법 2.5도 모호하다. 그림 2.6은 `1-2-3`에 대한 두 가지 구문 트리, 그림 2.7은 $1 + 2 * 3$에 대한 두 가지 트리를 보여준다. 구문 트리를 통해 식의 의미를 해석한다면, `1-2-3`의 두 구문 트리는 분명히 서로 다른 의미를 갖는다. $(1 - 2) - 3 = -4$와 $1 - (2 - 3) = 2$. 마찬가지로, $(1 + 2) * 3$과 $1 + (2 * 3)$은 다르다. 실제로 컴파일러는 구문 트리를 사용해 의미를 도출한다.

따라서 모호한 문법은 컴파일에 문제가 된다. 일반적으로 우리는 비모호 문법을 갖는 것이 바람직하다. 다행히도 모호한 문법을 비모호한 문법으로 바꿀수 있는 경우가 많다.

문법 2.5와 같은 언어를 받아들이는 비모호하지 않은 문법을 찾아보자. 먼저, 수학에서와 같이 `*`이 `+`보다 높은 *우선순위*를 가진다고 해석하고 싶다. 둘째, 각 연산자가 왼쪽 결합성을 부여하여, $1 - 2 - 3$를 $1 - (2 - 3)$ 대신 $(1 - 2) - 3$으로 해석하고 싶다. 이러한 해석을 위해 새로운 비터미널 기호를 도입하여 문법

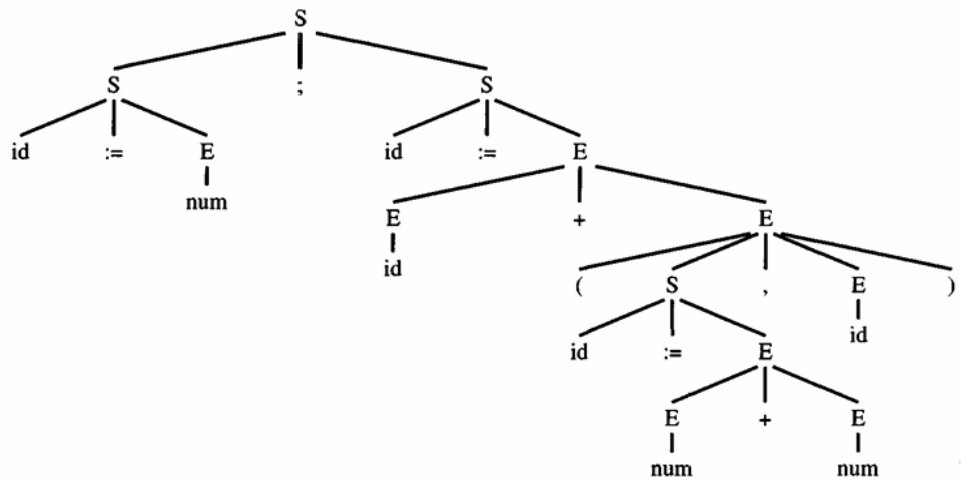


Figure 2.3: 구문 트리

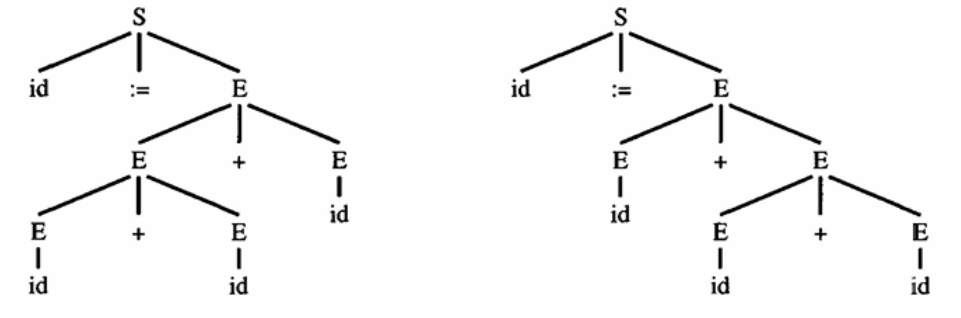


Figure 2.4: 문법 2.1로 같은 문장을 유도한 두 구문 트리

$$
\begin{aligned}
E &\to \texttt{id} \\
E &\to \texttt{num} \\
E &\to E * E \\
E &\to E / E \\
E &\to E + E \\
E &\to E - E \\
E &\to (E)
\end{aligned}
$$

Figure 2.5: 산술식 구문 문법

3.8을 정의한다.

문법 2.8에서 E, T, F는 각각 표현식(expression), 항(term), 인자(factor)를 뜻한다. 관례적으로 인자(factor)는 곱해지는 대상이고 항(term)은 더해지는 대상이다.

이 문법은 모호한 문법과 같은 언어를 받아들이지만, 이제 각 문장은 정확히 하나의 구문 트리를 가진다. 문법 2.8은 결코 그림 2.9와 같은 구문 트리를 만들지 않는다.

만약 *에 오른쪽 결합성을 부여하여 1 * 2 * 3을 1 * (2 * 3)으로 해석하고 싶다면 $T \to F * T$라고 쓰면 된다.

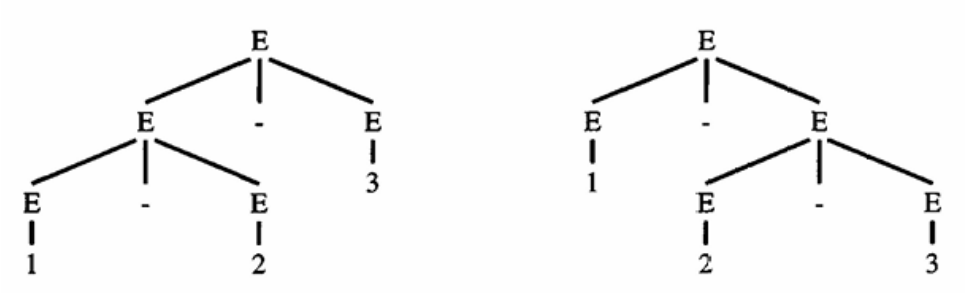


Figure 2.6: 문법 2.5로 같은 문장 1-2-3을 유도한 두 구문 트리

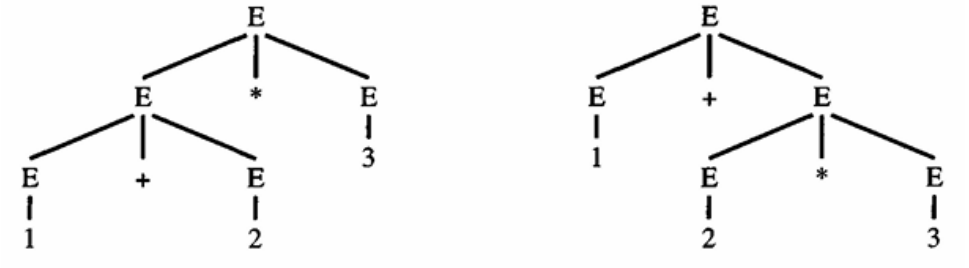


Figure 2.7: 문법 2.5로 같은 문장 1+2*3을 유도한 두 구문 트리

$E \to E + T$	$T \to T * F$	$F \to \texttt{id}$
$E \to E - T$	$T \to T/F$	$F \to \texttt{num}$
$E \to T$	$T \to F$	$F \to (\,E\,)$

Figure 2.8: 모호성을 해결한 산술식 구문 문법

보통 문법을 변형하여 모호성을 제거할 수 있다. 다만 어떤 언어는 모호한 문법은 있지만 모호하지 않는 문법은 없는 경우도 있다. 이런 언어는 프로그래밍 언어로는 부적합하다. 구문 구조가 모호하면 프로그램 작성과 이해에 문제를 일으킬 수 있기 때문이다.

파일 끝 표시(end-of-file marker)

구문 분석기는 `+`, `-`, `num` 같은 터미널 기호뿐 아니라 *파일 끝 표시*도 읽고 처리해야 한다. 파일 끝을 `$`로 나타낸다.

S가 문법의 시작 기호라고 하자. `$`가 S로 유도되는 구(S-phrase)의 다음에 와야 함을 나타내기 우해서, 새로운 시작 기호 S'와 새로운 생성 규칙 $S' \to S\$$를 보강하여 문법을 확장한다. 문법 2.5에서 시작 기호는 E이므로, 보강된 문법은 문법 2.10이 된다.

2.2 예측 구문 분석(Predictive Parsing)

어떤 문법은 **재귀 하향(recursive descent)**이라는 단순한 알고리즘으로 쉽게 분석할 수 있다. 문법 생성 규칙을 재귀 함수로 바로 구현할 수 있기 때문이다. 문법 2.11을 예시로 재귀 하향 구문 분석기를 작성하며 원리를 설명한다.

이 언어의 재귀 하향 구문 분석기는 비터미널마다 하나의 함수, 생성 규칙마다 하나의 절(clause)로 구현

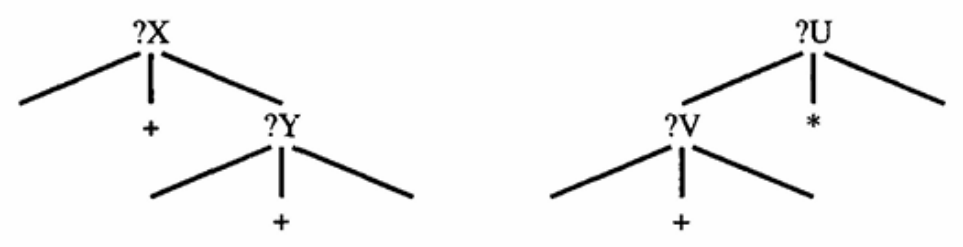


Figure 2.9: 문법 2.8로 유도할 수 없는 구문 트리들

$$
\begin{array}{lll}
S \to E\ \$ & & \\
E \to E + T & T \to T * F & F \to \texttt{id} \\
E \to E - T & T \to T / F & F \to \texttt{num} \\
E \to T & T \to F & F \to (\ E\ )
\end{array}
$$

Figure 2.10: 파일 끝 표시를 고려하도록 보강한 모호하지 않은 산술식 구문 문법

한다.

$$
\begin{array}{lll}
S \to \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S & L \to \texttt{end} & E \to \texttt{num} = \texttt{num} \\
S \to \texttt{begin}\ S\ L & L \to \texttt{;}\ S\ L & \\
S \to \texttt{print}\ E & &
\end{array}
$$

Figure 2.11: 파일 끝 표시를 고려하도록 보강한 모호하지 않은 산술식 구문 문법

```
datatype token = IF | THEN | ELSE | BEGIN | END | PRINT | SEMI | NUM | EQ

val tok = ref (getToken())
fun advance() = tok := getToken()
fun eat(t) = if (!tok=t) then advance() else error()

fun S() = case !tok
           of IF => (eat (IF); E(); eat (THEN); S(); eat (ELSE); S())
            | BEGIN => (eat (BEGIN); S(); L())
            | PRINT => (eat (PRINT); E())

and L() = case !tok
           of END => (eat (END))
            | SEMI => (eat (SEMI); S(); L())

and E() = (eat (NUM); eat (EQ); eat (NUM))
```

error와 getToken을 작성하면, 이 파싱 프로그램은 매우 잘 동작할 것이다.

문법 2.10에 이 방법을 시도해 본다:

```
fun S() = (E(); eat(EOF))
and E() = case !tok
 of ? => (E(); eat (PLUS); T())
  | ? => (E(); eat (MINUS); T())
  | ? => (T())
and T() = case !tok
  of ? => (T(); eat (TIMES); F())
   | ? => (T(); eat (DIV); F())
   | ? => (F())
and F() = case !tok
  of ID => (eat(ID))
   | NUM => (eat (NUM))
   | LPAREN => (eat (LPAREN); E(); eat (RPAREN))
```

여기에는 한 가지 충돌이 있다: E 함수가 어떤 절을 선택해야 할지 알 방법이 없다. 이러한 상황을 물음표로 표시했다. 문자열 $(1 * 2 - 3) + 4$와 $(1 * 2 - 3)$을 생각해 보면, 전자의 경우 E에 대한 초기 호출은 $E \to E + T$ 생성 규칙을 사용해야 하지만, 후자의 경우는 $E \to T$를 사용해야 한다.

순환 하강(recursive-descent) 파싱, 또는 예측(predictive) 파싱은 각 하위 표현식의 첫 번째 터미널 기호가 어떤 생성 규칙을 사용할지 선택하는 데 충분한 정보를 제공하는 문법에서만 작동한다. 이를 더 잘 이해하기 위해, FIRST 집합의 개념을 공식화한 다음, 충돌 없는 순환 하강 파서를 유도하는 알고리즘을 설명한다.

$$
\begin{array}{lll}
Z \to d & Y \to \epsilon & X \to Y \\
Z \to XYZ & Y \to c & X \to a
\end{array}
$$

Figure 2.12

어휘 분석기를 정규 표현식으로부터 구성할 수 있는 것처럼, 예측 파서를 만드는 파서 생성기 도구들이 있다. 하지만 도구를 사용할 것이라면, **??** 절에서 설명할 더 강력한 LR(1) 파싱 알고리즘에 기반한 도구를 사용하는 편이 좋다.

파서 생성기 도구를 사용하는 것이 때로 불편하거나 불가능할 수 있다. 예측 파싱의 장점은 알고리즘이 충분히 간단하여 자동화된 도구 없이도 직접 파서를 만들 수 있다는 것이다.

FIRST와 FOLLOW 집합 터미널 및 넌터미널 기호로 이루어진 문자열 γ가 주어졌을 때, $FIRST(\gamma)$는 γ로부터 유도될 수 있는 모든 문자열의 시작이 될 수 있는 터미널 기호들의 집합이다. 예를 들어, $\gamma = T * F$

라고 합시다. γ로부터 유도된 터미널 기호 문자열은 반드시 id, num, 또는 (로 시작해야 합니다. 따라서,

$$FIRST(T * F) = \{ \text{ id, num, (} \}$$

두 개의 다른 생성 규칙 $X \to \gamma_1$과 $X \to \gamma_2$가 동일한 좌변 기호(X)를 가지고 그들의 우변이 겹치는 FIRST 집합을 가진다면, 그 문법은 예측 파싱 방법을 사용하여 분석될 수 없다. 만약 어떤 터미널 기호 t가 $FIRST(\gamma_1)$과 $FIRST(\gamma_2)$에 모두 속해 있다면, 순환 하강 파서의 X 함수는 입력 토큰이 t일 때 무엇을 해야 할지 알 수 없을 것이다.

FIRST 집합의 계산은 매우 단순해 보인다. 만약 $\gamma = XYZ$라면, Y와 Z는 무시할 수 있고 $FIRST(X)$만이 중요한 것처럼 보인다. 하지만 문법 2.12을 보면, Y가 빈 문자열을 생성할 수 있고, 따라서 X도 빈 문자열을 생성할 수 있기 때문에, $FIRST(XYZ)$는 $FIRST(Z)$를 포함해야 한다는 것을 알 수 있다. 그러므로 FIRST 집합을 계산할 때, 어떤 기호가 빈 문자열을 생성할 수 있는지 추적해야 한다. 그런 기호들을 nullable하다고 한다. nullable한 기호 뒤에 무엇이 올 수 있는지를 추적해야 한다.

특정 문법에 대해, 터미널과 넌터미널로 이루어진 문자열 γ가 주어졌을 때:

- **nullable(X)**는 X가 빈 문자열을 유도할 수 있으면 참이다.
- **FIRST(γ)**는 γ로부터 유도된 문자열의 시작이 될 수 있는 터미널들의 집합이다.
- **FOLLOW(X)**는 X 바로 뒤에 올 수 있는 터미널들의 집합이다. 즉, Xt를 포함하는 유도가 존재하면 $t \in FOLLOW(X)$이다. 이것은 유도가 Y와 Z가 모두 ϵ으로 유도되는 $XYZt$를 포함할 경우 발생할 수 있다.

FIRST, FOLLOW, nullable의 정확한 정의는 다음 속성을 만족하는 가장 작은 집합들이다.

- 각 터미널 기호 Z에 대해, $FIRST[Z] = \{Z\}$.
- 각 생성 규칙 $X \to Y_1 Y_2 \dots Y_k$에 대해
  - 만약 $Y_1 \dots Y_k$가 모두 nullable이라면 (또는 $k = 0$이라면)
    - $nullable[X] = \text{true}$
  - 1부터 k까지의 i와 $i+1$부터 k까지의 j에 대해
    - 만약 $Y_1 \dots Y_{i-1}$가 모두 nullable이라면 (또는 $i = 1$이라면)
      - $FIRST[X] = FIRST[X] \cup FIRST[Y_i]$
    - 만약 $Y_{i+1} \dots Y_k$가 모두 nullable이라면 (또는 $i = k$이라면)
      - $FOLLOW[Y_i] = FOLLOW[Y_i] \cup FOLLOW[X]$
    - 만약 $Y_{i+1} \dots Y_{j-1}$가 모두 nullable이라면 (또는 $i + 1 = j$이라면)
      - $FOLLOW[Y_i] = FOLLOW[Y_i] \cup FIRST[Y_j]$

FIRST, FOLLOW, nullable을 계산하는 알고리즘 2.13은 바로 이 성질을 이용하여, 단순히 각 방정식을 할당문으로 바꾸고 반복하는 방식으로 설계한 것이다.

이 알고리즘을 효율적으로 만들려면 올바른 순서로 생성 규칙을 따라가는 것이 필요하다. 그리고, 동시에 계산하지 않고, 먼저 단독으로 nullable을 계산하고, 그 다음 FIRST, 그 다음 FOLLOW를 계산할 수 있다.

집합에 대한 방정식에서 그 집합을 계산하는 알고리즘을 설계한 것과 비슷하게, ϵ-클로저 계산 알고리즘을 설계했었다. 그리고 이 알고리즘과 같이 고정점에 도달할 때까지 반복하는 기법(repeat until no change)은 컴파일러의 백엔드에서 최적화를 위한 데이터 흐름 분석에 적용 가능하다.

초기화:
모든 비단말 X에 대해 FIRST$[X] \leftarrow \emptyset$, FOLLOW$[X] \leftarrow \emptyset$, nullable$[X] \leftarrow$ false
각 터미널 기호 Z에 대해 FIRST$[Z] \leftarrow \{Z\}$

repeat
  foreach *생성 규칙* $X \to Y_1 Y_2 \cdots Y_k$ **do**
    if $Y_1 \cdots Y_k$*가 모두 nullable* **또는** $k = 0$ **then**
      nullable$[X] \leftarrow$ true
    end
    for $i \leftarrow 1$ **to** k **do**
      if $Y_1 \cdots Y_{i-1}$*이 모두 nullable* **또는** $i = 1$ **then**
        FIRST$[X] \leftarrow$ FIRST$[X] \cup$ FIRST$[Y_i]$
      end
      if $Y_{i+1} \cdots Y_k$*가 모두 nullable* **또는** $i = k$ **then**
        FOLLOW$[Y_i] \leftarrow$ FOLLOW$[Y_i] \cup$ FOLLOW$[X]$
      end
      for $j \leftarrow i + 1$ **to** k **do**
        if $Y_{i+1} \cdots Y_{j-1}$*이 모두 nullable* **또는** $i + 1 = j$ **then**
          FOLLOW$[Y_i] \leftarrow$ FOLLOW$[Y_i] \cup$ FIRST$[Y_j]$
        end
      end
    end
  end
until *FIRST, FOLLOW, nullable이 변하지 않을 때까지*;

Figure 2.13: FIRST, FOLLOW, nullable 집합 계산 알고리즘

이 알고리즘을 문법 2.12에 적용할 수 있다. 초기값은 다음과 같다.

	nullable	**FIRST**	**FOLLOW**
X	아니오		
Y	아니오		
Z	아니오		

첫 번째 반복에서, $a \in FIRST[X]$, Y는 nullable, $c \in FIRST[Y]$, $d \in FIRST[Z]$, $d \in FOLLOW[X]$, $c \in FOLLOW[X]$, $d \in FOLLOW[Y]$임을 알게 된다. 따라서:

	nullable	**FIRST**	**FOLLOW**
X	아니오	a	c,d
Y	예	c	d
Z	아니오	d	

두 번째 반복에서, X가 nullable, $c \in FIRST[X]$, $\{a, c\} \subseteq FIRST[Z]$, $\{a, c, d\} \subseteq FOLLOW[X]$, $\{a, c, d\} \subseteq FOLLOW[Y]$임을 알게 된다. 따라서:

	nullable	FIRST	FOLLOW
X	예	a,c	a,c,d
Y	예	c	a,c,d
Z	아니오	a,c,d	

세 번째 반복에서는 새로 추가되는 정보가 없으므로 알고리즘을 종료한다.

FIRST 관계를 기호 문자열로 일반화하는 것이 유용하다:

- $FIRST(X\gamma) = FIRST[X]$, nullable[X]가 아닐 경우
- $FIRST(X\gamma) = FIRST[X] \cup FIRST(\gamma)$, nullable[X]일 경우

비슷하게, 문자열 γ의 각 기호가 nullable일 때 그 문자열 γ를 nullable하다고 말합니다.

예측 파서 구축하기

	a	c	d
X	$X \to a$ $X \to Y$	$X \to Y$	$X \to Y$
Y	$Y \to \epsilon$	$Y \to \epsilon$ $Y \to c$	$Y \to \epsilon$
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$

Figure 2.14: 문법 2.12에 대한 예측 파싱 테이블

순환 하강 파서의 넌터미널 X에 대한 파싱 함수는 각 X-생성 규칙에 대한 절을 가지며, 입력의 다음 토큰 T를 기반으로 이 절들 중 하나를 선택해야 한다. 각 (X,T)에 대해 올바른 생성 규칙을 선택할 수 있다면, 순환 하강 파서를 작성할 수 있다. 필요한 모든 정보는 넌터미널 X와 터미널 T로 인덱싱된 2차원 생성 규칙 테이블로 인코딩될 수 있다. 이를 예측 파싱 테이블이라고 한다.

이 테이블을 구축하려면, 각 $T \in FIRST(\gamma)$에 대해 테이블의 X행, T열에 생성 규칙 $X \to \gamma$를 추가한다. 또한, γ가 nullable하다면, 각 $T \in FOLLOW[X]$에 대해 X행, T열에 이 생성 규칙을 추가한다.

그림 2.14는 문법 2.12에 대한 예측 파서를 보여준다. 하지만 일부 항목에는 하나 이상의 생성 규칙이 포함되어 있다! 중복된 항목의 존재는 예측 파싱이 문법 2.12에서 작동하지 않을 것임을 의미한다.

문법을 더 자세히 살펴보면, 모호하다는 것을 알 수 있다. 문장 d는 다음과 같은 여러 파스 트리로 유도할 수 있다:

```
Z          Z
|         /|\
d        X Y Z
        ... |
            d
```

모호한 문법은 항상 예측 파싱 테이블에 중복된 항목을 만든다. 만약 문법 2.12의 언어를 프로그래밍 언어로 사용해야 한다면, 모호하지 않은 문법을 찾아야 할 것이다.

예측 파싱 테이블에 중복된 항목이 없는 문법을 $LL(1)$이라고 한다. 이것은 좌측에서 우측으로(Left-to-right) 파싱, 최좌측 유도(Leftmost-derivation), 1-기호 예측(1-symbol lookahead)을 의미한. 순환 하강(예측) 파서는 분명히 입력을 왼쪽에서 오른쪽으로 한 번의 패스로 검사한다. 일부 파싱 알고리즘은 그렇지 않지만, 컴파일러에는 일반적으로 유용하지 않다. 예측 파서가 넌터미널을 우변으로 확장하는 순서(즉, 순환 하강 파서가 넌터미널에 해당하는 함수를 호출하는 순서)는 바로 최좌측 유도가 넌터미널을 확장하는 순서이다. 그리고 순환 하강 파서는 입력의 다음 토큰 하나만 보고 작업을 수행하며, 절대로 한 토큰 이상을 미리 보지 않는다.

FIRST 집합의 개념을 일반화하여 문자열의 첫 k개 토큰을 기술하고, 행이 넌터미널이고 열이 모든 k개 터미널 시퀀스인 $LL(k)$ 파싱 테이블을 만들 수 있다. 이것은 테이블이 너무 크기 때문에 거의 사용되지 않지만, 때때로 직접 순환 하강 파서를 작성할 때 한 토큰 이상을 미리 봐야 할 필요가 있다. $LL(2)$ 파싱 테이블로 파싱 가능한 문법을 $LL(2)$ 문법이라고 하며, $LL(3)$ 등도 마찬가지이다. 모든 $LL(1)$ 문법은 $LL(2)$ 문법이며, $LL(2)$ 문법은 $LL(3)$ 문법이고, 계속 이어진다. 어떤 모호한 문법도 임의의 k에 대해 $LL(k)$가 될 수 없다.

좌측 재귀 제거 (Eliminating Left Recursion) 문법 2.10에 대한 예측 파서를 만들어 보자. 두 생성 규칙

$$E \to E + T$$
$$E \to T$$

는 $FIRST(T)$에 있는 어떤 토큰이든 $FIRST(E+T)$에도 있을 것이기 때문에, $LL(1)$ 파싱 테이블에 중복된 항목을 야기할 것이 분명하다. 문제는 E가 E-생성 규칙의 우변 첫 번째 기호로 나타난다는 점입니다; 이것을 *좌측 재귀(left recursion)*라고 한다. 좌측 재귀가 있는 문법은 $LL(1)$이 될 수 없다.

좌측 재귀를 제거하기 위해, 우측 재귀를 사용하여 다시 작성한다. 새로운 넌터미널 E'을 도입하고 다음과 같이 바꾼다.

$$E \to TE'$$
$$E' \to +TE'$$
$$E' \to \epsilon$$

이것은 원래의 두 생성 규칙과 동일한 (T와 +에 대한) 문자열 집합을 유도하지만, 이제는 좌측 재귀가 없다. 일반적으로, α가 X로 시작하지 않는 $X \to X\gamma$와 $X \to \alpha$ 형태의 생성 규칙이 있을 때, 이것이 $\alpha\gamma*$ 형태, 즉 하나의 α 다음에 0개 이상의 γ가 오는 문자열을 유도한다. 따라서 우측 재귀를 사용하여 정규 표현식을

다시 작성할 수 있다.

$$\begin{pmatrix} X \to X\gamma_1 \\ X \to X\gamma_2 \\ X \to \alpha_1 \\ X \to \alpha_2 \end{pmatrix} \Rightarrow \begin{pmatrix} X \to \alpha_1 X' \\ X \to \alpha_2 X' \\ X' \to \gamma_1 X' \\ X' \to \gamma_2 X' \\ X' \to \epsilon \end{pmatrix}$$

이 변환을 문법 2.10에 적용하면 문법 2.15가 된다. 예측 파서를 만들기 위해, 먼저 nullable, FIRST, FOLLOW를 계산한다(표 2.16). 문법 2.15에 대한 예측 파서는 표 2.17에 나와 있습니다.

$S \to E\$$ $T \to FT'$ $F \to \text{id}$
$E \to TE'$ $T' \to *FT'$ $F \to \text{num}$
$E' \to +TE'$ $T' \to /FT'$ $F \to (E)$
$E' \to -TE'$ $T' \to \epsilon$
$E' \to \epsilon$

Figure 2.15: 좌측 재귀를 없앤 문법

	nullable	**FIRST**	**FOLLOW**
S	아니오	(id num	
E	아니오	(id num	) $
E'	예	+ -	) $
T	아니오	(id num	) + - $
T'	예	* /	) + - $
F	아니오	(id num	) * / + - $

Figure 2.16: 문법 3.15에 대한 Nullable, FIRST, FOLLOW.

	id	**(**	**+**	*****	**)**	**$**
S	$S \to E\$$	$S \to E\$$				
E	$E \to TE'$	$E \to TE'$				
E'			$E' \to +TE'$		$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	$T \to FT'$				
T'			$T' \to \epsilon$	$T' \to *FT'$	$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \text{id}$	$F \to (E)$				

Figure 2.17: 문법 2.15에 대한 예측 파싱 테이블. num, /, -에 대한 열은 테이블의 다른 열과 유사하므로 생략한다.

좌인수분해 (Left Factoring) 우리는 좌측 재귀가 예측 파싱을 방해하고, 그것이 제거될 수 있다는 것을 보았다. 동일한 넌터미널에 대한 두 생성 규칙이 동일한 기호로 시작할 때 비슷한 문제가 발생한다. 예를 들어,

```
S -> if E then S else S
S -> if E then S
```

이런 경우, 문법을 *좌인수분해(left factor)*할 수 있다. 즉, 끝 부분("else S" 또는 ϵ)을 나타내는 새로운 넌터미널 X를 만든다.

$$S \rightarrow \text{if } E \text{ then } SX$$
$$X \rightarrow \epsilon$$
$$X \rightarrow \text{else } S$$

그 결과, 예측 파서에 문제를 일으키지 않는 생성 규칙들이 된다.

오류 복구 (Error Recovery) 예측 파싱 테이블을 구하면, 순환 하강 파서를 쉽게 작성할 수 있다. 다음은 문법 2.15에 대한 파서의 일부 코드다.

```
and T() =
  case !tok
   of ID => (F(); T'())
    | NUM => (F(); T'())
    | LPAREN => (F(); T'())
    | _ => error()


and T'() = case !tok
            of PLUS => ()
             | TIMES => (eat(TIMES); F(); T'())
             | RPAREN => ()
             | EOF => ()
             | _ => error()
```

LL(1) 파싱 테이블의 T행, x열에 있는 빈 항목은 파싱 함수 T()가 토큰 x를 예상하지 않는다는 것을 나타낸다. 이것은 구문 오류가 될 것이다. 오류는 어떻게 처리해야 할까? 예외를 발생시키고 파싱을 중단하는 것이 안전하지만, 이것은 사용자에게 그다지 친절하지 않다. 오류 메시지를 출력하고 오류로부터 복구하여 동일한 컴파일에서 다른 구문 오류를 찾을 수 있도록 하는 것이 더 좋다.

구문 오류는 입력 토큰 문자열이 해당 언어의 문장이 아닐 때 발생한다. 오류 복구는 그 토큰 문자열과 유사한 어떤 문장을 찾는 방법으로, 토큰을 삭제, 교체 또는 삽입함으로써 진행될 수 있다. 예를 들어, T에 대한 오류 복구는 num 토큰을 삽입함으로써 진행될 수 있다. 실제 입력을 조정할 필요는 없다. num이 거기에 있었다고 가정하고, 메시지를 출력하고, 정상적으로 반환하는 것으로 충분하다.

```
and T() = case !tok
           of ID => (F(); T'())
            | NUM => (F(); T'())
            | LPAREN => (F(); T'())
            | _ => print("id, num, 또는 왼쪽 괄호가 필요합니다")
```

토큰 삽입을 통한 오류 복구는 약간 위험하다. 왜냐하면 오류가 연쇄적으로 다른 오류를 발생시키면 프로세스가 무한히 반복될 수 있기 때문이다. 삭제를 통한 오류 복구는 더 안전하다. 왜냐하면 파일의 끝에 도달하면 루프가 결국 종료되어야 하기 때문이다. 삭제를 통한 간단한 복구는, FOLLOW 집합에 속한 토큰이 나올 때까지 토큰을 건너뛰는 방식으로 작동한다. 예를 들어, T'에 대한 오류 복구는 다음과 같이 작동할 수 있다:

```
and T'() = case !tok
            of PLUS => ()
             | TIMES => (eat(TIMES); F(); T'())
             | RPAREN => ()
             | EOF => ()
             | _ => (print "+, *, 오른쪽 괄호, 또는 파일의 끝이 필요합니다";
                     skipto[PLUS, TIMES, RPAREN, EOF])

and skipto(stop) =
     if member(!tok, stop) then ()
     else (eat(!tok); skipto(stop))
```

순환 하강 파서의 오류 복구 메커니즘은 단 하나의 토큰이 잘못된 위치에 있어 발생하는 긴 연쇄적인 오류 복구 메시지를 피하기 위해 때로는 시행착오를 통해 조정되어야 한다.

2.3 LR 파싱

$LL(k)$ 파싱 기술의 약점은 생성 규칙의 우변에서 첫 k개의 토큰만 보고 어떤 생성 규칙을 사용할지 예측해야 한다는 것이다. 더 강력한 기술인 $LR(k)$ 파싱은 해당 생성 규칙의 전체 우변에 해당하는 입력 토큰들(그리고 그 너머의 k개 입력 토큰들까지)을 모두 볼 때까지 결정을 미룰 수 있다.

LR(k)는 좌에서 우로 파싱(Left-to-right parse), 최우측 유도(Rightmost-derivation), k-토큰 예측(k-token lookahead)을 의미한다. 최우측 유도를 사용하는 것이 이상하게 보일 수 있는데, 이것이 어떻게 좌에서 우로 파싱과 양립할 수 있는가? 그림 2.18은 새로운 시작 생성 규칙 $S' \rightarrow S\$$가 추가된 문법 2.1을 사용하여 프로그램 `a := 7; b := c + (d:= 5+ 6, d)`를 LR 파싱하는 과정을 보여준다.

파서는 스택과 입력을 가진다. 입력의 첫 k개 토큰이 예측(lookahead) 토큰이 된다. 스택의 내용과 예측 토큰을 기반으로 파서는 두 종류의 동작을 수행한다.

- Shift(이동): 첫 번째 입력 토큰을 스택의 맨 위로 옮긴다.
- Reduce(축소): 문법 규칙 $X \rightarrow ABC$를 선택하고, 스택의 맨 위에서 C, B, A를 pop한 뒤, X를 스택에 push한다.

초기에는 스택이 비어 있고 파서는 입력의 시작 부분에 있다. 파일 끝 마커 $를 이동(shift)하는 동작을 **accepting(승인)**이라고 하며, 파싱이 성공했고 파서를 종료한다.

그림 2.18에서는 매 단계 후의 스택과 입력이 방금 수행된 동작의 표시와 함께 나타나 있다. 스택과 입력의 연결은 항상 최우측 유도가 된다. 보기 좋도록, 그림 2.18은 입력 문자열의 최우측 유도를 거꾸로 보여준다.

Figure 2.18: 문장의 이동-축소 파싱. 스택의 숫자 아래첨자는 DFA 상태 번호이다; 표 2.19 참조.

Stack	Input	**Action**
1	a := 7 ; b := c + (d := 5 + 6 , d) $	shift
1 id_4	:= 7 ; b := c + (d := 5 + 6 , d) $	shift
1 id_4 :=6	7 ; b := c + (d := 5 + 6 , d) $	shift
1 id_4 :=6 num_{10}	; b := c + (d := 5 + 6 , d) $	reduce E → num
1 id_4 :=6 E_{11}	; b := c + (d := 5 + 6 , d) $	reduce S → id := E
1 S_2	; b := c + (d := 5 + 6 , d) $	shift
1 S_2 ;3	b := c + (d := 5 + 6 , d) $	shift
1 S_2 ;3 id_4	:= c + (d := 5 + 6 , d) $	shift
1 S_2 ;3 id_4 :=	c + (d := 5 + 6 , d) $	shift
1 S_2 ;3 id_4 :=6	c + (d := 5 + 6 , d) $	shift
1 S_2 ;3 id_4 :=6 id_{20}	+ (d := 5 + 6 , d) $	reduce E → id
1 S_2 ;3 id_4 :=6 E_{11}	+ (d := 5 + 6 , d) $	shift
1 S_2 ;3 id_4 :=6 E_{11} +16	(d := 5 + 6 , d) $	shift
1 S_2 ;3 id_4 :=6 E_{11} +16 (8	d := 5 + 6 , d) $	shift
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4	:= 6 , d) $	shift
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6	5 + 6 , d) $	shift
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 num_{10}	, d) $	reduce E → num
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 E_{11}	, d) $	shift
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 E_{11} +16	d) $	shift
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 E_{11} +16 num_{10}	) , d) $	reduce E → num
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 E_{11} +16 E_{17}	, d) $	reduce E → E + E
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 E_{11}	, d) $	reduce S → id := E
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 S_{12}	, d) $	shift
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 S_{12} ,18	d) $	shift
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 S_{12} ,18 id_{20}	) $	reduce E → id
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 S_{12} ,18 E_{21}	) $	shift
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 S_{12} ,18 E_{21})22	$	reduce E → (S,E)
1 S_2 ;3 id_4 :=6 E_{11} +16 E_{17}	$	reduce E → E + E
1 S_2 ;3 id_4 :=6 E_{11}	$	reduce S → id := E
1 S_2 ;3 S_5	$	reduce S → S ; S
1 S_2	$	accept

LR 파싱 엔진 LR 파서는 언제 이동하고 언제 축소해야 하는지 어떻게 아는가? 결정적 유한 오토마타(DFA)를 사용한다! DFA는 입력에 적용되지 않고(유한 오토마타는 문맥 자유 문법을 파싱하기에는 너무 약하다) 스택에 적용된다. DFA의 간선들은 스택에 나타날 수 있는 기호들(터미널 및 넌터미널)로 레이블이 지정된다. 표 2.19는 문법 2.1에 대한 전이 테이블이다.

전이 테이블의 요소들은 네 종류의 동작으로 레이블이 지정된다.

상태	id	num	print	;	,	+	:=	(	)	$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4						s6							
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9	s20	s10						s8				g15	g14
10	r5		r5	r5	r5				r5	r5			
11			r2	r2	s16				r2				
12			s3	s18									
13			r3	r3						r3			
14				s19					s13				
15				r8					r8				
16	s20	s10						s8				g17	
17			r6	r6	s16				r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

Figure 2.19: 문법 2.1에 대한 LR 파싱 테이블.

0. $S' \rightarrow S$ \$
1. $S \rightarrow (\ L\ )$
2. $S \rightarrow x$
3. $L \rightarrow S$
4. $L \rightarrow L\ ,\ S$

Figure 2.20

- **sn**: 상태 n으로 이동(Shift);
- **gn**: 상태 n으로 가기(Goto);
- **rk**: 규칙 k로 축소(Reduce);
- **a**: 승인(Accept);
- **오류** (테이블의 빈 항목으로 표시).

파싱에서 이 테이블을 사용하려면, 이동(shift) 및 가기(goto) 동작을 DFA의 간선으로 취급하고 스택을 스캔한다. 예를 들어, 스택이 `id := E`이면 DFA는 상태 1에서 4, 6, 11로 이동한다. 다음 입력 토큰이 세미콜론이면, 상태 11의 ";" 열은 규칙 2로 축소하라고 지시한다. 문법의 두 번째 규칙은 $S \rightarrow id := E$이므로, 상위 세 개의 토큰이 스택에서 pop되고 S가 push된다.

상태 11에서 "+"에 대한 동작은 이동(shift)이다. 따라서 다음 토큰이 +였다면, 입력에서 소모되어 스택에 push되었을 것이다.

각 토큰에 대해 스택을 다시 스캔하는 대신, 파서는 각 스택 요소에 대해 도달한 상태를 기억할 수 있다. 그러면 파싱 알고리즘은 다음과 같다.

- 스택의 최상위 상태와 입력 기호를 찾아 파서 동작을 찾는다.
- 다음의 동작을 찾으면,
  - **Shift(n)**: 입력을 한 토큰 전진시키고, 스택에 n을 push한다.
  - **Reduce(k)**: 규칙 k의 우변에 있는 기호의 수만큼 스택을 pop한다. 규칙 k의 좌변 기호가 X라고 하면, pop한 결과 스택의 최상위 상태에서 X를 찾아 "goto n"을 얻는다. 스택의 최상위에 n을 push한다.
  - **Accept**: 파싱을 중단하고 성공을 보고한다.
  - **Error**: 파싱을 중단하고 실패를 보고한다.

LR(0) PARSER GENERATION $LR(k)$ 파서는 스택의 내용과 입력의 다음 k개 토큰을 사용하여 어떤 동작을 취할지 결정한다. 표 2.19는 하나의 예측 기호(lookahead)를 사용하는 것을 보여준다. $k = 2$인 경우, 테이블은 모든 두 토큰 시퀀스에 대한 열을 가진다. 실제로는 컴파일에 $k > 1$은 사용되지 않는다. 테이블이 거대해지고, 대부분의 프로그래밍 언어가 $LR(1)$ 문법으로 기술될 수 있기 때문이다.

$LR(0)$ 문법은 예측(lookahead) 없이 오직 스택만 보고 이동/축소 결정을 내리며 파싱할 수 있는 문법이다. 이 문법 클래스로 커버되지 못하는 프로그래밍 언어들이 있어 실용적이라 말할 수 없지만, $LR(0)$ 파싱 테이블을 구성하는 알고리즘은 $LR(1)$ 파서 구성 알고리즘에 대한 좋은 입문이 된다.

$LR(0)$ 파서 생성을 설명하기 위해 문법 2.20을 사용한다. 이 파서가 어떻게 동작할지 미리 생각해보자. 초기에는 스택이 비어 있고, 입력은 완전한 S-문장에 $를 붙인 것이므로, S' 규칙의 우변이 입력으로 주어진다. 우리는 이것을 $S' \to .S\$$로 표시하며, 점은 파서의 현재 위치를 나타낸다.

입력이 S로 시작하는 이 상태에서는, S-생성 규칙의 가능한 모든 우변으로 시작한다는 의미이다. 우리는 이것을 다음과 같이 나타낸다:

1

$$\boxed{\begin{aligned} S' &\to \cdot\, S\,\$ \\ S &\to \cdot\, x \\ S &\to \cdot\, (\,L\,) \end{aligned}}$$

이것을 상태 1이라고 부르자. 문법 규칙과 그 우변의 위치를 나타내는 점(dot)을 결합한 것을 *항목(item)* (구체적으로 $LR(0)$ 항목)이라고 한다. 상태는 바로 항목들의 집합이다.

이동(Shift) 동작. 상태 1에서, x를 입력 받고 이동(shift)하면 스택에 x를 넣는다. 이 상황을, $S \to x$ 생성 규칙에서 점을 x 뒤로 이동시켜 표현한다. 규칙 $S' \to .S\$$와 $S \to .(L)$은 입력 x를 받는 이 동작과 무관하므로 무시한다. 그 결과 상태 2에 도달한다.

2

$$\boxed{S \to \mathtt{x}\,\cdot}$$

또는 상태 1에서 왼쪽 괄호를 입력 받고 이동(shift)할 수 있다. 상태 1의 세 번째 항목에서 점을 괄호 뒤로 옮기면 $S \to (.L)$이 되며, 여기서 스택의 맨 위에는 왼쪽 괄호가 있어야 하고, 입력은 L에 의해 유도된

어떤 문자열로 시작하고, 그 뒤에 오른쪽 괄호가 있을 것이다. 이제 입력은 어떤 토큰으로 시작할 수 있는가? 우리는 모든 L-생성 규칙을 항목 집합에 포함시킴으로써 알아낸다. 그리고, 그 L-항목들 중 하나에서 점이 S 바로 앞에 있으므로, 모든 S-생성 규칙을 포함시켜야 한다.

3

$S \to (\cdot\, L\,)$

$L \to \cdot\, L\,, S$

$L \to \cdot\, S$

$S \to \cdot\, (\,L\,)$

$S \to \cdot\, \texttt{x}$

가기(Goto) 동작. 상태 1에서, S 넌터미널에 의해 유도된 어떤 토큰 문자열을 파싱하는 효과를 생각해 보자. 이것은 x나 왼쪽 괄호가 이동(shift)되고, 최후에 S-생성 규칙의 축소(reduction)가 뒤따를 때 발생할 것이다. 그 생성 규칙의 모든 우변 기호들이 pop되고, 파서는 상태 1에서 S에 대한 goto 동작을 실행할 것이다. 이 효과는 상태 1의 첫 번째 항목에서 점을 S 뒤로 옮겨 상태 4를 만듦으로써 시뮬레이션할 수 있다:

4

$S' \to S \cdot \$$

축소(Reduce) 동작. 상태 2에서 항목의 끝에 점이 있다. 이것은 해당하는 생성 규칙($S \to x$)의 우변 전체가 스택의 맨 위에 놓여 있고 축소될 준비가 되어 있다는 것을 의미한다. 그런 상태에서 파서는 축소 동작을 수행할 수 있다.

앞서 우리가 설명한 상태 전이에서 기본적으로 수행한 연산들은 **closure**(I)와 **goto**(I, X)라고 부른다. 여기서 I는 항목들의 집합이고 X는 문법 기호(터미널 또는 넌터미널)이다. **Closure**는 점이 넌터미널의 왼쪽에 있을 때 항목 집합에 더 많은 항목을 추가하고, **goto**는 모든 항목에서 점을 기호 X 뒤로 옮긴다.

```
Closure(I) =
  repeat
    for I 안의 임의의 아이템 A -> α .X β 에 대해
      for 임의의 생성규칙 X -> γ 에 대해
        I <- I ∪ { X -> .γ }
  until I 가 더 이상 변하지 않을 때.
  return I

Goto(I, X) =
  J를 공집합으로 설정
  for I 안의 임의의 아이템 A -> α .X β 에 대해
    J에 A -> α X. β 를 추가
  return Closure(J)
```

이제 LR(0) 파서 구성 알고리즘을 설명한다. 먼저, 보조 시작 생성 규칙 $S' \rightarrow S\$$로 문법을 확장한다. 상태들의 집합을 T, 간선들의 집합을 E라고 하며, E에는 터미널에 대한 Shift와 비터미널에 대한 Goto 간선이 포함된다. 알고리즘은 T와 E에 새 상태 또는 간선을 반복적으로 추가하고, 더 이상 추가할 것이 없을 때(고정점에 도달하면) 종료한다.

T를 $\{\text{Closure}(\{S' \rightarrow \cdot S\$\})\}$로 초기화한다.
E를 $\emptyset$로 설정한다.
repeat
  for T의 각 상태 I에 대해
    for I의 각 항목 $A \rightarrow \alpha \cdot X\beta$에 대해
      J <- $Goto(I, X)$
      T <- $T \cup \{J\}$
      E <- $E \cup \{I \xrightarrow{X} J\}$
until E와 T가 이번 반복에서 변하지 않을 때

기호 \$에 대해서는 $Goto(I, \$)$를 계산하지 않고, 대신 accept 동작을 만들 것이다. 문법 2.20에 대해 이것은 그림 5.10에 설명되어 있다. 이제 LR(0) 축소 동작의 집합 R을 계산할 수 있다:

R <- $\emptyset$
for T의 각 상태 I에 대해
  for I 안의 각 항목 $A \rightarrow a \cdot$ 에 대해
    R <- $R \cup \{(I,\ A \rightarrow a)\}$

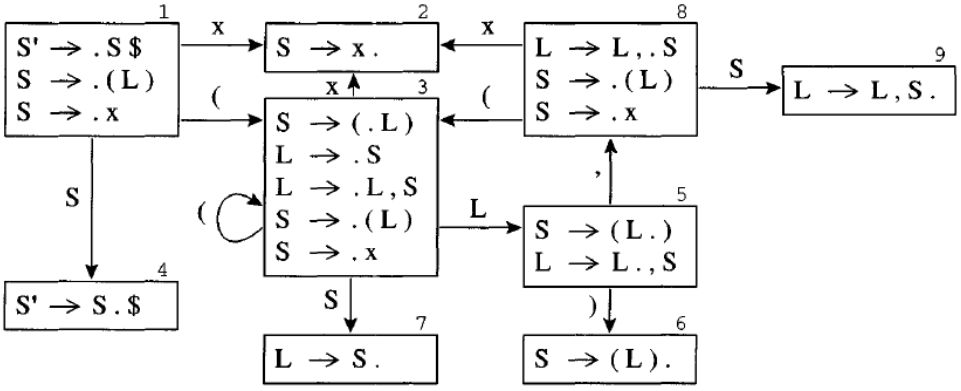


Figure 2.21: 문법 2.20에 대한 LR(0) 상태들.

이제 이 문법에 대한 파싱 테이블을 표 2.22로 구성할 수 있다. 각 간선 $I \xrightarrow{X} J$에 대해, X가 터미널이면 테이블의 (I, X) 위치에 shift J 동작을 넣는다. X가 넌터미널이면 (I, X) 위치에 goto J를 넣는다. $S' \rightarrow S.\$$ 항목을 포함하는 각 상태 I에 대해 (I, \$)에 accept 동작을 넣는다. 마지막으로, 점이 끝에 있는 (n번) 생성 규칙 η인 항목 $A \rightarrow \gamma.$를 포함하는 상태에 대해, 모든 토큰 Y에 대해 (I, Y)에 reduce η (reduce n) 동작을 넣는다.

$LR(0)$은 예측(lookahead)을 사용하지 않으므로, 각 상태에 대해 이동하거나 축소하는 단일 동작만 필요하다. 둘 다 하지는 않는다. 실제로는, 어떤 상태로 이동해야 하는지 알아야 하기 때문에, 상태 번호로 시작하는 행과 문법 기호로 시작하는 열을 가진다.

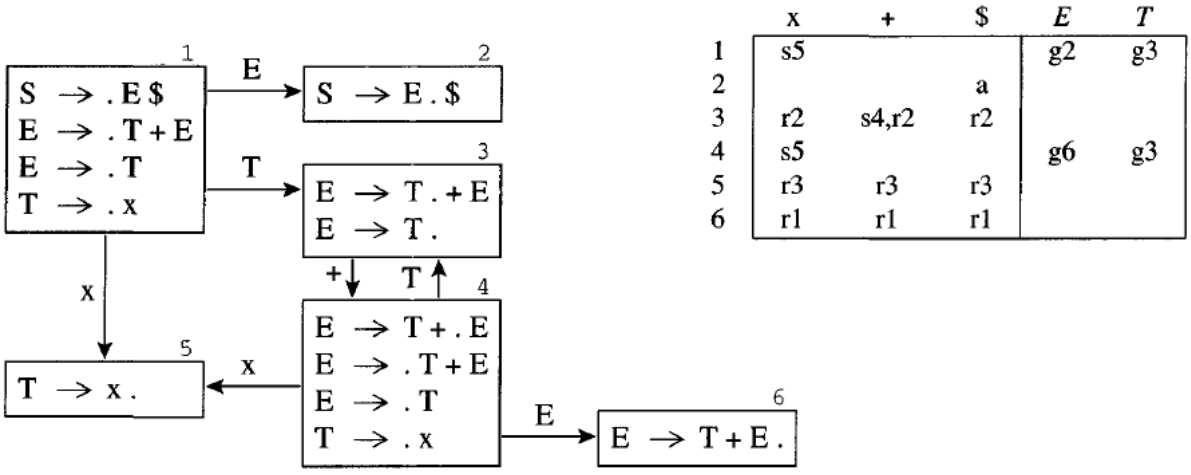
상태	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

Figure 2.22: 문법 2.20에 대한 LR(0) 파싱 테이블.

$$S \to E\$ \quad E \to T + E$$
$$E \to T \quad T \to x$$

Figure 2.23: LR(0) 파싱 테이블을 만들기 위한 문법 예제

SLR PARSER GENERATION 문법 2.23에 대한 LR(0) 파싱 테이블을 만들어 보자. LR(0) 상태와 파싱 테이블은 그림 2.24에 나와 있다. 상태 3에서, 기호 +에 대해 중복된 항목이 있다. 파서는 상태 4로 이동해야 하거나, 생성 규칙 2로 축소해야 한다. 충돌이 발생했다. 따라서, 이 문법이 $LR(0)$이 아님을 나타낸다. $LR(0)$ 파서를 사용하여 파싱될 수 없는 문법이다. 더 정교한 파싱 알고리즘이 필요하다.



	x	+	$	E	T
1	s5			g2	g3
2			a		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		

Figure 2.24: 문법 2.23에 대한 LR(0) 상태 및 파싱 테이블.

SLR(Simple LR)은 LR(0)보다 나은 파서를 구성하는 간단한 방법이다. SLR 파서 구성은 LR(0)과 거의 동일하지만, FOLLOW 집합이 나타내는 곳에만 축소 동작을 테이블에 넣는다는 점이 다르다.

SLR 테이블에 축소 동작을 넣는 알고리즘은 다음과 같다.

```
R <- ∅
for T의 각 상태 I에 대해
  for I 안의 각 항목 A → a·에 대해
    for 각 토큰 X ∈ FOLLOW(A)에 대해
      R <- R ∪ {(I, X, A → a)}
```

동작 (I, X, $A \to \alpha$)는 상태 I에서 예측 기호 X를 만났을 때, 파서가 규칙 $A \to \alpha$로 축소할 것임을

나타낸다. 따라서, 문법 2.23에 대해 우리는 동일한 $LR(0)$ 상태 다이어그램(그림 2.24)을 사용하지만, 그림 2.25에 표시된 대로 SLR 테이블에는 더 적은 개수의 축소 동작을 넣는다.

SLR 문법 클래스는 SLR 파싱 테이블에 충돌(중복 항목)이 없는 문법들이다. 문법 2.23은 이 클래스에 속하며, 많은 유용한 프로그래밍 언어 문법도 마찬가지이다.

상태	x	+	$	E	T
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

Figure 2.25: 문법 3.23에 대한 SLR 파싱 테이블.

LR(1) 항목과 LR(1) 파싱 테이블 SLR보다 훨씬 정교한 방식이 $LR(1)$ 파싱 알고리즘이다. 문맥 자유 문법으로 구문을 기술할 수 있는 대부분의 프로그래밍 언어는 LR(1) 문법을 가진다.

$LR(1)$ 파싱 테이블을 구성하는 알고리즘은 LR(0)과 유사하지만, 항목의 개념이 더 정교하다. LR(1) 항목은 문법 생성 규칙, 우변에서의 위치(점으로 표현), 그리고 예측 기호(lookahead symbol)로 구성된다. 항목 $(A \to \alpha.\beta, x)$이 의도하는 아이디어는, 스택의 최상위에 시퀀스 α가 있고, βx로부터 유도될 수 있는 입력 문자열이 있다는 것이다.

$LR(1)$ 상태는 $LR(1)$ 항목들의 집합이며, 추가된 예측을 고려하여 $LR(1)$에 대한 Closure와 Goto 연산을 다음과 같이 정의한다.

```
Closure(I) =
  repeat
    for I 안의 임의의 아이템 (A → α · Xβ, z) 에 대해
      for 임의의 생성규칙 X → γ 에 대해
        for 임의의 기호 w ∈ FIRST(βz) 에 대해
          I <- I ∪ { (X → ·γ, w) }
  until I 가 더 이상 변하지 않을 때
  return I

Goto(I, X) =
  J <- ∅
  for I 안의 임의의 아이템 (A → α · Xβ, z) 에 대해
    J에 (A → αX · β, z) 를 추가
  return Closure(J)
```

시작 상태는 항목 $(S' \to .S\$, ?)$의 클로저이며, 여기서 예측 기호 ?는 중요하지 않다. 왜냐하면 파일 끝 마커는 절대로 이동(shift)되지 않기 때문이다. 축소 동작은 다음 알고리즘에 의해 선택된다:

```
R <- ∅
for T의 각 상태 I에 대해
```

`for` I 안의 각 항목 $(A \to \alpha \cdot,\ z)$에 대해
  `R <-` $R \cup \{(I,\ z,\ A \to \alpha)\}$

동작 $(I, z, A \to \alpha)$는 상태 I에서 예측 기호 z를 만났을 때, 파서가 규칙 $A \to \alpha$로 축소할 것임을 나타낸다.

문법 2.26은 SLR이 아니지만, $LR(1)$ 문법 클래스에 속한다. 그림 2.27은 이 문법에 대한 $LR(1)$ 상태들을 보여준다.

이 그림에서 동일한 생성 규칙을 가지되 룩어헤드(lookahead)가 서로 다른 아이템들이 있는 경우(아래 왼쪽처럼), 오른쪽과 같이 약식으로 표기했다.

$S' \to \cdot S\,\$$	?
$S \to \cdot V = E$	\$
$S \to \cdot E$	\$
$E \to \cdot V$	\$
$V \to \cdot x$	\$
$V \to \cdot * E$	\$
$V \to \cdot x$	$=$
$V \to \cdot * E$	$=$

$S' \to \cdot S\,\&$	?
$S \to \cdot V = E$	\$
$S \to \cdot E$	\$
$E \to \cdot V$	\$
$V \to \cdot x$	\$, $=$
$V \to \cdot * E$	\$, $=$

이 상태 그래프로부터 도출한 LR(1) 구문 분석 표는 표 2.28a이다.

- 생성 규칙의 끝에 점($\cdot$)이 놓인 경우, $LR(1)$ 테이블에는 해당 생성 규칙에 대한 축소 동작이 있으며, 상태 번호에 해당하는 행과 항목의 예측 기호에 해당하는 열에 배치한다. 그림 2.27의 상태 3에서처럼, 여기서 점은 생성 규칙 $E \to V$의 끝에 있다. 이 경우 예측 기호는 \$이다. 상태 3과 예측 기호 \$로 특정되는 위치에 3번 생성 규칙을 축소하는 동작 r3을 넣는다.

- 점이 터미널 기호나 넌터미널의 왼쪽에 놓인 경우, $LR(0)$ 테이블에서와 마찬가지로 $LR(1)$ 파싱 테이블에 해당하는 이동(goto) 또는 가기(shift) 동작이 있다.

LALR(1) 파싱 테이블 $LR(1)$ 파싱 테이블은 많은 상태를 가지며 매우 클 수 있다. 예측 집합을 제외하고 항목이 동일한 두 상태를 병합하여 더 작은 테이블을 만들 수 있다. 이렇게 만든 파서를 LALR(1) 파서(Look-Ahead LR(1))라고 한다. 예를 들어, 문법 2.26에 대한 $LR(1)$ 파서(그림 2.27)의 상태 6과 13의 항목들은 예측 집합을 무시하면 동일하다. 또한, 상태 7과 12는 예측 기호를 제외하고 동일하며, 상태 8과 11, 상태 10과 14도 마찬가지이다. 이 상태 쌍들을 합하면 표 2.28b에 표시된 LALR(1) 파싱 테이블이 된다.

0. $S' \to S\$$ 3. $E \to V$
1. $S \to V = E$ 4. $V \to x$
2. $S \to E$ 5. $V \to *E$

Figure 2.26: C 언어의 표현식, 변수, 포인터 역참조(* 연산자)의 본질을 포착하는 문법.

일부 문법의 경우, LALR(1) 테이블은 LR(1) 테이블에는 없는 축소-축소 충돌을 포함할 수 있다. 그러한 경우를 제외하면, $LALR(1)$ 파싱 테이블이 상태 수가 훨씬 적을 수 있으므로 $LR(1)$ 테이블보다 표현하는 데 더 적은 메모리가 필요하다는 점이다.

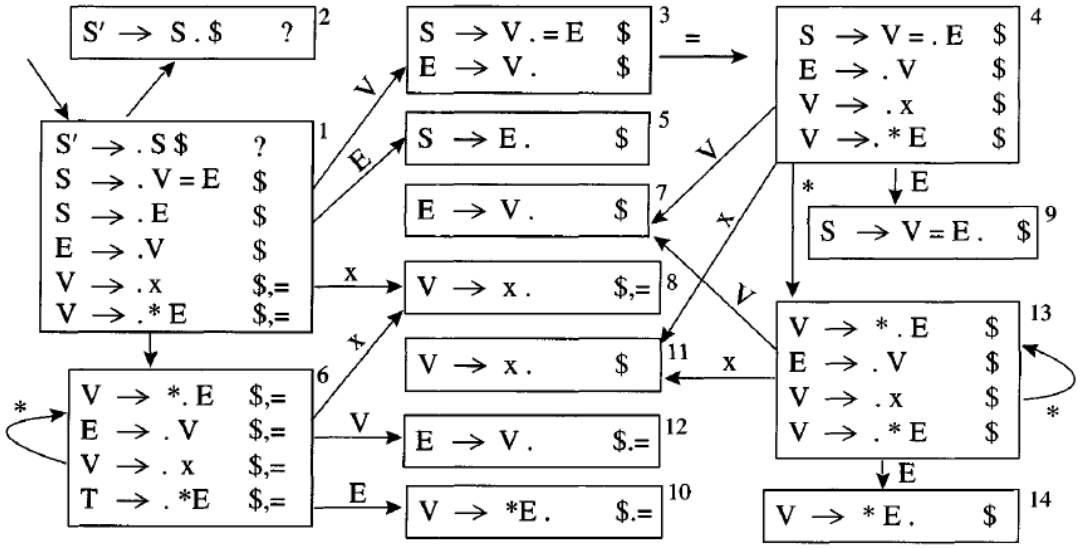


Figure 2.27: 문법 2.26에 대한 LR(1) 상태들.

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

(a) LR(1)

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s8	s6				g9	g7
5				r2			
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

(b) LALR(1)

Figure 2.28: 문법 2.26에 대한 LR(1) 및 LALR(1) 파싱 테이블.

문법 클래스의 계층 구조 $LALR(1)$ 파싱 테이블에 충돌이 없는 문법을 $LALR(1)$ 문법이라고 한다. 모든 SLR 문법은 $LALR(1)$이지만, 그 반대는 성립하지 않는다. 그림 2.29는 여러 문법 클래스 간의 관계를 보여준다.

일반적으로 프로그래밍 언어는 LALR(1) 문법을 가지며, LALR(1) 문법을 위한 많은 파서 생성기 도구가 있다. 이런 이유로 $LALR(1)$은 프로그래밍 언어 및 자동 파서 생성기의 표준이 되었다.

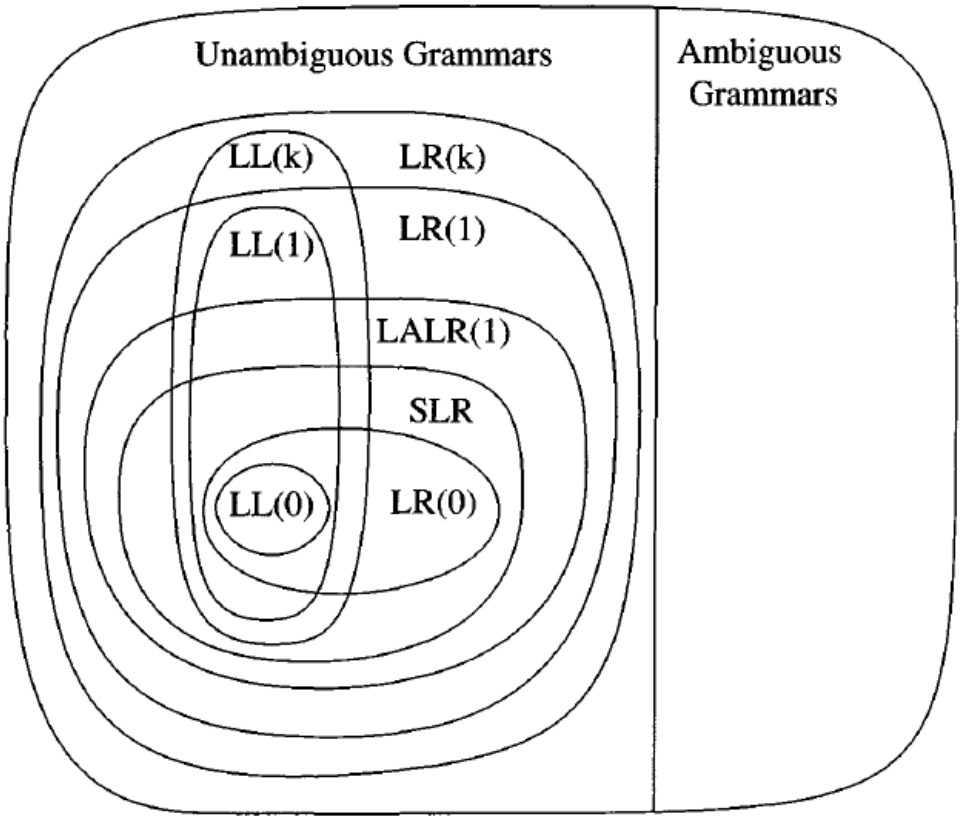


Figure 2.29: 문법 클래스의 계층 구조.

모호한 문법의 LR PARSING 많은 프로그래밍 언어에는 다음과 같은 문법 규칙이 있다:

```
S -> if E then S else S
S -> if E then S
S -> other
```

이는 다음과 같은 프로그램을 허용한다:

```
if a then if b then s1 else s2
```

이 프로그램은 두 가지 방식으로 이해될 수 있다:

1. if a then { if b then s1 else s2 }
2. if a then { if b then s1 } else s2

대부분의 프로그래밍 언어에서, else는 가장 최근의 가능한 then과 일치하도록 정하므로, 해석 (1)을 선택하다. 하지만, LR 파싱 테이블에는 이동-축소 충돌이 있을 것이다:

```
S -> if E then S .   [lookahead: else]
S -> if E then S . else S
```

이동(Shifting)은 해석 (1)에 해당하고 축소(reducing)는 해석 (2)에 해당한다.

이 모호성은 보조 넌터미널 M(일치된 문장)과 U(일치되지 않은 문장)를 도입하여 제거할 수 있다.

```
S -> M | U
M -> if E then M else M | other
U -> if E then S | if E then M else U
```

하지만 문법을 다시 작성하는 대신, 문법을 변경하지 않고 이동-축소 충돌을 해결하면 좋겠다. 파싱 테이블을 구성할 때 이 충돌은 해석 (1)을 선호하므로 이동(shifting)으로 해결되어야 한다.

적절하게 이동-축소 충돌을 이동 또는 축소 쪽으로 해결함으로써 모호한 문법을 사용하는 것이 종종 가능하다. 하지만 이 기술은 드물게, 그리고 잘 알려진 경우(여기에 설명된 매달린-else와 연산자 우선순위 등)에만 사용하는 것이 가장 좋다. 대부분의 이동-축소 충돌, 그리고 아마도 모든 축소-축소 충돌은 파싱 테이블을 주먹구구식으로 해결해서는 안 된다. 그것들은 잘못 명시된 문법이라는 증상이며, 모호성을 제거함으로써 해결되어야 한다.

2.4 연습문제

다음 문법에 대해 LR(0) 아이템 집합을 상태로 하고, 심볼(Shift/Goto 액션)이 표시된 에지로 하는 DFA를 작성하시오. 이 DFA로 부터 액션 테이블과 Goto 테이블을 작성하시오.

$$
\begin{aligned}
0 &: S \rightarrow E\ \$ \\
1 &: E \rightarrow id \\
2 &: E \rightarrow id\ (\ E\ ) \\
3 &: E \rightarrow E\ +\ id
\end{aligned}
$$

위 LR(0) 파서 오토마톤으로 id (id) + id를 파싱해복, 각 세부 단계를 설명하시오.

Chapter 3

의미 분석: 타입 시스템

3.1 타입 시스템 개요

타입 시스템(type system)은 타입 추론 규칙(inference rule)과 공리(axiom)들의 집합으로 구성되며, 이를 통해 타입이 잘 매겨졌는지 판정한다(typing judgment). 다음과 같은 형태로 타입 판정을 기술한다.

$$\Gamma \vdash p : \tau$$

이 판정은 프로그램(또는 프로그램 구문) p가 식별자 타입 매김(identifier typing) Γ에 대해 타입 τ를 가진다는 것을 의미한다. 여기서 식별자 타입 매김 Γ는 식별자들을 타입에 대응시키는 함수로, p에 등장하는 자유 식별자들의 타입을 지정해준다. 공리이거나 앞선 판정들로부터 타입 추론 규칙을 통해 새로운 판정을 도출한다. 일련의 판정들 중 마지막에 위치하면 타입 시스템으로부터 도출된 판정이라고 한다.

예를 들어, 정수 값을 가지는 표현식에 대한 단순한 타입 시스템을 생각해 보자. 이 타입 시스템은 다음 세 가지 규칙을 포함할 수 있다. 첫째, 모든 정수 리터럴 i가 타입 `int`를 가진다는 공리

$$\Gamma \vdash i : int$$

둘째, $\Gamma(x) = \tau$이면 자유 식별자 x의 타입이 τ임을 주는 추론 규칙

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

셋째, 두 표현식 e, e'가 각각 타입 `int`라면, $e + e'$ 역시 타입 `int`라는 추론 규칙

$$\frac{\Gamma \vdash e : int \quad \Gamma \vdash e' : int}{\Gamma \vdash e + e' : int}$$

추론 규칙에서, 가로선 위의 판정들은 가설(hypotheses)이고, 가로선 아래의 판정은 결론(conclusion)이다. 예를 들어 $\Gamma(z) = int$라면

$$\Gamma \vdash z + 1 : int$$

라는 판정을 얻을 수 있다. 이 경우 $z+1$은 Γ에 대해 잘 타입이 매겨졌다고(well typed) 하며, 타입 `int`를 가진다고 말한다. 반면 $\Gamma(z) = bool$이라면, 위 판정은 시스템으로부터 도출되지 않으므로 $z+1$은 Γ에 대해 잘 타입이 매겨지지 않았다고 한다.

3.2 WHILE 시맨틱스와 타입 시스템

$$v ::= n \mid b \quad (n \in Z,\ b \in \{\mathsf{True}, \mathsf{False}\})$$

$$e ::= n \mid b \mid x \mid e + e \mid e - e \mid e < e \mid e == e \mid e \wedge e \mid e \vee e \mid \neg e$$

$$c ::= \texttt{skip} \mid c; c \mid x = e \mid \texttt{read}(x) \mid \texttt{write}(e) \mid \texttt{if}\ (e)\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ (e)\ c$$

판단의 형태

$$Env \vdash e \Rightarrow v \quad \text{(식 평가)} \qquad Env \vdash c \Rightarrow Env' \quad \text{(문장 실행)}$$

3.2.1 동적 의미 (Big-step Operational Semantics)

식 평가 규칙 $Env \vdash e \Rightarrow v$

$$\frac{}{Env \vdash n \Rightarrow n}\ \textsc{Const(int)} \qquad \frac{}{Env \vdash b \Rightarrow b}\ \textsc{Const(bool)} \qquad \frac{Env(x) = v}{Env \vdash x \Rightarrow v}\ \textsc{Var}$$

$$\frac{Env \vdash e_1 \Rightarrow n_1 \qquad Env \vdash e_2 \Rightarrow n_2}{Env \vdash e_1 + e_2 \Rightarrow n_1 + n_2}\ \textsc{Add} \qquad \frac{Env \vdash e_1 \Rightarrow n_1 \qquad Env \vdash e_2 \Rightarrow n_2}{Env \vdash e_1 - e_2 \Rightarrow n_1 - n_2}\ \textsc{Sub}$$

$$\frac{Env \vdash e_1 \Rightarrow n_1 \qquad Env \vdash e_2 \Rightarrow n_2}{Env \vdash e_1 < e_2 \Rightarrow (n_1 < n_2)}\ \textsc{Lt} \qquad \frac{Env \vdash e_1 \Rightarrow v_1 \qquad Env \vdash e_2 \Rightarrow v_2}{Env \vdash e_1 == e_2 \Rightarrow (v_1 = v_2)}\ \textsc{Eq}$$

$$\frac{Env \vdash e_1 \Rightarrow b_1 \qquad Env \vdash e_2 \Rightarrow b_2}{Env \vdash e_1 \wedge e_2 \Rightarrow (b_1 \wedge b_2)}\ \textsc{And} \qquad \frac{Env \vdash e_1 \Rightarrow b_1 \qquad Env \vdash e_2 \Rightarrow b_2}{Env \vdash e_1 \vee e_2 \Rightarrow (b_1 \vee b_2)}\ \textsc{Or}$$

$$\frac{Env \vdash e \Rightarrow b}{Env \vdash \neg e \Rightarrow \neg b}\ \textsc{Not}$$

문장 실행 규칙 $Env \vdash c \Rightarrow Env'$

$$\frac{}{Env \vdash \texttt{skip} \Rightarrow Env}\ \textsc{Skip} \qquad \frac{Env \vdash e \Rightarrow v}{Env \vdash x = e \Rightarrow Env\{x \mapsto v\}}\ \textsc{Assign}$$

$$\frac{Env \vdash c_1 \Rightarrow Env' \qquad Env' \vdash c_2 \Rightarrow Env''}{Env \vdash c_1; c_2 \Rightarrow Env''}\ \textsc{Seq}$$

$$\frac{Env \vdash e \Rightarrow \mathsf{true} \qquad Env \vdash c_1 \Rightarrow Env'}{Env \vdash \texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Rightarrow Env'}\ \textsc{If-True}$$

$$\frac{Env \vdash e \Rightarrow \mathsf{false} \qquad Env \vdash c_2 \Rightarrow Env'}{Env \vdash \texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \Rightarrow Env'}\ \textsc{If-False} \qquad \frac{Env \vdash e \Rightarrow \mathsf{false}}{Env \vdash \texttt{while}\ (e)\ c \Rightarrow Env}\ \textsc{While-False}$$

$$\frac{Env \vdash e \Rightarrow \mathsf{true} \qquad Env \vdash c \Rightarrow Env' \qquad Env' \vdash \texttt{while}\ (e)\ c \Rightarrow Env''}{Env \vdash \texttt{while}\ (e)\ c \Rightarrow Env''}\ \textsc{While-True}$$

$$\frac{\text{입력으로 } v\text{를 받음}}{Env \vdash \texttt{read}(x) \Rightarrow Env\{x \mapsto v\}}\ \textsc{Read} \qquad \frac{Env \vdash e \Rightarrow v}{Env \vdash \texttt{write}(e) \Rightarrow Env}\ \textsc{Write}$$

비정상(오류) 경우 조건식이 bool 값으로 평가되지 않는 경우 등은 규칙이 적용되지 않아 실행 결과 환경이 정의되지 않는(오류) 것으로 본다. 이 관례는 슬라이드의 "otherwise error!" 설명과 일치한다.

실행 예 $Env = \{x \mapsto 1\}$라 하고, x := x 1 을 계산해보자.

$$\dfrac{\dfrac{\dfrac{\dfrac{Env(x) = 1}{Env \vdash x \Rightarrow 1}\ \textsc{Var} \qquad \dfrac{}{Env \vdash 1 \Rightarrow 1}\ \textsc{Const(int)}}{Env \vdash x + 1 \Rightarrow 2}\ \textsc{Add}}{Env \vdash x := x + 1 \Rightarrow Env\{x \mapsto 2\}}\ \textsc{Cmd}$$

3.2.2 타입 시스템

타입과 타입 환경

$$\tau ::= \mathsf{int} \mid \mathsf{bool} \qquad \Gamma : \mathrm{Var} \to \{\mathsf{int}, \mathsf{bool}\} \qquad \Gamma[x \mapsto \tau]$$

판단: $\Gamma \vdash e : \tau$ (식의 타입), $\Gamma \vdash c$ (문장 *well-formed*)

식의 타입 규칙 $\Gamma \vdash e : \tau$

$$\frac{}{\Gamma \vdash n : \mathsf{int}}\ \text{T-Const(int)} \qquad \frac{}{\Gamma \vdash b : \mathsf{bool}}\ \text{T-Const(bool)} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{T-Var}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}\ \text{T-Add} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 - e_2 : \mathsf{int}}\ \text{T-Sub}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 < e_2 : \mathsf{bool}}\ \text{T-Lt} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 == e_2 : \mathsf{bool}}\ \text{T-Eq}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1 \wedge e_2 : \mathsf{bool}}\ \text{T-And} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1 \vee e_2 : \mathsf{bool}}\ \text{T-Or} \qquad \frac{\Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash \neg e : \mathsf{bool}}\ \text{T-Not}$$

문장의 타입(정합성) $\Gamma \vdash c$

$$\frac{}{\Gamma \vdash \texttt{skip}}\ \text{T-Skip} \qquad \frac{\Gamma(x) = \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x = e}\ \text{T-Assign} \qquad \frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1; c_2}\ \text{T-Seq}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2}\ \text{T-If} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash c}{\Gamma \vdash \texttt{while}\ (e)\ c}\ \text{T-While} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash \texttt{read}(x)}\ \text{T-Read}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{write}(e)}\ \text{T-Write}$$

프로그램 수준(선언 + 본문) 선언부에서 변수의 타입을 모아 Γ를 구성하고, 그 환경에서 본문 문장이 정합적인지 검사한다:

$$\frac{\text{decls} \Rightarrow \Gamma \qquad \Gamma \vdash c}{\vdash\ \text{decls};\ c}$$

타입 규칙 적용 사례 $\Gamma = \{x : int\}$라 하고, x := x 1 의 타입을 분석해보자.

$$\dfrac{\dfrac{\dfrac{\Gamma(x) = int}{\Gamma \vdash x : int}\ \text{T-Var} \qquad \dfrac{}{\Gamma \vdash 1 : int}\ \text{T-Const(INT)}}{\Gamma \vdash x + 1 : int}\ \text{T-Add}}{\Gamma \vdash x := x + 1}\ \text{T-Assign}$$

타입 건전성(개념) 정적 타입 검사에 통과한 프로그램은 실행 중 "정수 연산에 비정수 사용, 조건식의 비불리언 사용, 선언되지 않은 변수 사용"과 같은 오류가 발생하지 않도록 보장한다(루프 종료, 0으로 나눔 등은 별도).

환경 Evn의 타입에 해당하는 타입 환경 Γ을 $\vdash Env : \Gamma$라 표시하고, 다음과 같이 정의한다.

- 모든 i에 대하여 $\emptyset \vdash v_i : \tau_i$이면, $\vdash \{x_1 \mapsto v_1, \cdots, x_n \mapsto v_n\} : \{x_1 : \tau_1, \cdots, x_n : \tau_n\}$이다.

Theorem 3.2.1 (식에 대한 타입 건전성). $\vdash Env : \Gamma$, $\Gamma \vdash e : \tau$, $Env \vdash e \Rightarrow v$ *이면,* $\Gamma \vdash v : \tau$*이다.*

Theorem 3.2.2 (커맨드 타입 건전성). $\vdash Env : \Gamma$, $Env \vdash c \Rightarrow Env'$ *이면,* $\vdash Env' : \Gamma$

3.3 정보 흐름을 통제하는 타입 시스템(Information-Flow Type System)

정보흐름 보안(Information-Flow Security)의 핵심 목표는 민감한 데이터가 허락받지 않은 사용자에게 새어나가지 않도록 프로그램의 동작을 통제하는 것이다. 접근 제어나 암호화와 같은 전통적인 접근 방식은 데이터의 저장이나 전송은 보호할 수 있지만, 프로그램 실행 과정에서 발생하는 숨은 정보 흐름(데이터 흐름이나 제어 흐름을 통한 간접적 유출)까지는 완전히 막아 주지 못한다. 이런 한계를 보완하기 위해 프로그래밍 언어 차원에서 정보흐름 정책을 지정하고, 정적 분석으로 이를 보장하는 타입 시스템이 제안되었다.

Denning의 격자 모델: Denning [Den76]은 Bell-LaPadula 모델 [BL73]을 확장하여 의 격자(lattice) 모델을 제시하였다. 이 모델에서 보안 정책은 격자 $(SC, \leq)$로 정의된다. 클래스를 격자로 정의하여, 정보가 한 방향으로만 흐를 때 허용하는 정책을 따른다.

- SC: 보안 클래스의 유한 집합
- $\leq$: 보안 클래스들 사이의 부분 순서

예를 들어, 낮음(L)과 높음(H)을 원소로 하는 기밀성 클래스와 신뢰함(T)과 신뢰하지 않음(U)을 원소로 하는 무결성 클래스를 정의할 수 있다.

- 기밀성(Secrecy) 클래스: $SC = \{\ L,\ H\ \}, L \leq H$
- 무결성(Integrity) 클래스: $SC = \{\ T,\ U\ \}, T \leq U$

또한 기밀성과 무결성을 조합한 클래스(예: HT)도 가능하다.

다만, Denning이 제시한 당시의 보안 분석 방법은 정확성(건전성)이 수학적으로 증명되지 않았다는 한계가 있었다.

타입 시스템으로의 발전: 이후 연구에서는 Denning의 아이디어를 정보 흐름을 통제하는 타입 시스템으로 정리하여, 타입이 잘 매겨진 모든 프로그램은 허락되지 않은 관찰자(또는 공격자)가 높은 수준의 정보에 의해 영향을 받지 않는다는 성질, 즉 비간섭성(Noninterference)을 보일 수 있음을 증명하였다 [VIS96].

- 다시 말해, 타입 규칙으로 정보흐름 정책을 정의하고,
- 타입 건전성 증명으로 정보 유출이 일어나지 않음을 보장한다.

비간섭성은, 쉽게 말하면 비밀 정보가 프로그램의 실행 결과에 전혀 흔적을 남기지 않는다는 뜻이다.

예를 들어, 프로그램이 학생의 시험 점수(비밀)와 학번(공개)* 입력받아 학번만 출력한다면 점수가 몇 점이든 출력 결과에는 차이가 없으므로 정보가 유출되지 않는다. 반면 프로그램이 평균 점수까지 출력한다면, 공격자는 그 값을 이용해 간접적으로 개별 점수에 대한 단서를 얻을 수 있으므로 비간섭성이 보장되지 않는다.

Volpano 등이 제안한 이 타입 시스템 [VIS96]은 WHILE 프로그램을 대상으로 정의되었으며, 이후에는 고차 함수를 포함한 함수형 프로그램을 다루는 연구도 진행되었다 [PS02].

이러한 접근은 프로그래밍 언어 차원에서 이루어지는 정보흐름 보안(Language-based Information-Flow Security) 기법이라고 부른다.

프로그래밍 언어 기반 정보 흐름 보안: 프로그래밍 언어 차원에서, 즉 고도화된 타입 시스템을 통해 정보흐름 보안을 제공하는 방법에 대한 연구는 이후 더욱 확장되었다.

- 언어의 표현력: 절차, 함수, 예외, 객체
- 동시성: 비결정성, 스레드 동시성, 분산
- 은닉 채널: 종료 채널, 타이밍 채널, 확률적 채널
- 보안 정책

이와 같은 주제별 연구 동향은 Sabelfeld와 Myers 논문 [SM03]에 잘 정리되어 있다.

3.3.1 격자 기반 정보 흐름 모델

격자(lattice) 모델은 Bell과 LaPadula 모델 [BL73]의 확장이다. 이 모델에서 정보 흐름 정책은 격자 $(SC, \leq)$로 정의되며, 여기서 SC는 보안 클래스들의 유한 집합이고 $\leq$는 그들 사이의 부분 순서를 나타낸다. SC에는 기밀성(secrecy) 클래스, 예를 들어 낮음(L)과 높음(H), 그리고 무결성(integrity) 클래스, 예를 들어 신뢰됨(T)과 신뢰되지 않음(U)이 포함될 수 있으며, $L \leq H$, $T \leq U$ 관계가 성립한다. 또한 HT와 같은 이들의 조합도 가능하다.

모든 프로그램 변수 x는 보안 클래스 $class(x)$를 가진다고 가정하며, 이는 정적으로 결정 가능하고 실행 중에는 변하지 않는다고 가정한다. 변수 x에서 변수 y로 정보 흐름이 발생한다면, 이는 $x \leq y$일 때만 허용된다.

모든 프로그래밍 구성 요소는 고유한 인증 조건(certification condition)을 갖는다. 프로그램의 소스 코드 구조에 따라 각 명령문이나 표현식이 어떤 보안 클래스 조건을 만족하는지 정의한다. 이러한 조건들을 통해 명시적으로 또는 암묵적 정보흐름을 통제한다.

명시적으로 정보흐름을 통제하는 조건의 예로, 대입문 $y := x$을 들 수 있다. 이때 $\underline{x} \leq \underline{y}$이며, 이는 x의 보안 클래스에서 y의 보안 클래스로의 명시적 흐름(explicit flow)을 허용함을 의미한다. 반대로 y에서 x로의 흐름은 허용되지 않는다.

반면, 조건문(if)이나 반복문(while)에서는 암묵적으로 정보흐름을 통제한다. 예를 들어, 조건문의 분기에는 항상 조건식(guard)으로부터의 암묵적 흐름이 존재한다. 다음 문장을 보자.

$$\texttt{if } x > y \texttt{ then } z := w \texttt{ else } i := i + 1$$

여기서는 x와 y에서 z와 i로의 암묵적 흐름이 발생한다. 따라서 이 문장의 인증 조건은

$$\underline{x} \oplus \underline{y} \leq \underline{z} \otimes \underline{i}$$

가 되며, 여기서 $\oplus$와 $\otimes$는 각각 최소 상한(least upper bound)과 최대 하한(greatest lower bound) 연산자를 나타낸다.

격자(lattice) 성질 덕분에 이러한 조건들은 단순한 속성 문법(attribute grammar)으로, 합성 속성(synthesized attributes)만을 사용하여 강제할 수 있다.

3.3.2 정보흐름 타입 시스템 살펴보기

3.1절에서 전통적인 타입 시스템을 살펴보았다. 우리가 다루는 보안 흐름 타입 시스템(secure flow type system)도 마찬가지로 타입과 타입 추론 규칙으로 구성된다. 그러나 이 경우 규칙들은 데이터 타입의 호환성을 보장하는 대신, 정보 흐름의 보안(security of information flow)을 보장한다. 즉, 이 규칙들을 통해 블록 구조(block-structured)를 가지는 순차 실행 언어(deterministic language) 내의 표현식과 명령문에 대해 보안 흐름 판정을 내릴 수 있다.

안전한 정보흐름 타입(Secure Flow Types)

우리의 타입 시스템에서 타입은 두 가지 수준으로 구분된다. 첫 번째 수준은 데이터 타입(data type)으로, τ로 표기하며 이는 보안 클래스 집합 SC의 원소이다. SC가 부분 순서 $\leq$에 의해 정렬되어 있다고 가정한다.

두 번째 수준은 구문 단위 타입(phrase type)으로, ρ로 표기한다. 구문 단위 타입에는 데이터 타입(즉, 표현식에 부여되는 타입), $\tau\ var$ 형태의 변수 타입, 그리고 $\tau\ cmd$ 형태의 명령문 타입이 포함된다.

- 예상할 수 있듯, $\tau\ var$ 타입의 변수는 보안 클래스가 τ 이하인 정보를 저장한다.
- 보다 특이하게, 명령문 c가 $\tau\ cmd$ 타입을 갖는다는 것은 c 안의 모든 대입문이 보안 클래스가 τ 이상인 변수에 대해서만 수행된다는 것을 보장한다는 뜻이다. 이는 암묵적 정보 흐름(implicit flow)에서도 보안을 보장하기 위해 필요한 격리 속성(confinement property)이다.

우리는 부분 순서 $\leq$를 서브타입(subtype) 관계로 확장하며, 이를 $\subseteq$로 표기한다. 서브타입 관계는 명령문의 타입에 대해서는 반단조적(antimonotonic) 또는 반공변적(contravariant)이다. 즉, 만약 $\tau \subseteq \tau'$이면

$$\tau'\ cmd \subseteq \tau\ cmd$$

가 성립한다.

마지막으로, 일반적인 타입 시스템과 마찬가지로, 만약 $\rho \subseteq \rho'$라면 타입이 ρ인 구문은 ρ' 타입으로 형변환(type coercion)될 수 있다는 규칙이 존재한다.

안전한 정보흐름 타입 매김 규칙

타입 매김 규칙(typing rule)은 격자 모델(lattice model)의 인증 조건(certification rule)과 마찬가지로 명시적(explicit) 및 암묵적(implicit) 정보 흐름을 모두 안전하게 보장한다.

명시적으로 정보 흐름이 이루어지는 대입문에 대한 타입 매김 규칙은 다음과 같다.

$$\frac{\Gamma \vdash e : \tau\ var \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e := e' : \tau\ cmd}$$

이 규칙의 의미는, e'에서 e로의 명시적 흐름이 보안상 안전하려면 e'와 e가 동일한 보안 수준을 가져야 한다는 것이다. 이는 τ가 두 가설(hypothesis) 모두에 나타나도록 한 점에서 드러난다. 다만, e'에서 e로의 상향(upward) 흐름은 여전히 허용된다. 예를 들어 $e : H\ var$이고 $e' : L$이라면, 서브타입 매김(subtyping)을 통해 e'의 타입을 H로 올려서(coercion) $\tau = H$로 규칙을 적용할 수 있다.[1]

명시적 정보 흐름 타입 매김 예제 x가 H, y가 L일 때 높은 수준의 정보를 낮은 수준의 정보를 저장하는 y := x 프로그램을 허용하면 안된다. 정보 흐름 타입 시스템을 적용하면 아래와 같이 타입을 매길 수 없다. 이러한 'Bad' 프로그램을 방지할 수 있다.

$$\dfrac{\dfrac{\Gamma(y) = L\ \text{var}}{\lambda; \Gamma \vdash y : L\ \text{var}}\ \textsc{Var} \qquad \dfrac{\dfrac{\Gamma(x) = H\ \text{var} \neq L\ \text{var}}{\lambda; \Gamma \not\vdash x : L\ \text{var}}\ \textsc{var}}{\lambda; \Gamma \vdash x : L}\ \textsc{r\_val}}{\lambda; \{x : H\ \text{var}, y : L\ \text{var}\} \vdash y := x : L\ \text{cmd}}\ \textsc{Assign}$$

반면에, x가 L, y가 H일 때 y := x 프로그램은 허용해야 된다. 왜냐하면, 낮은 수준의 정보를 높은 수준의 변수에 저장할 수 있기 때문이다. 정보 흐름 타입 시스템을 적용하면 (Subtype)으로 x의 낮은 수준 L 정보를 H로 격상시켜 타입을 매길 수 있다.

$$\dfrac{\dfrac{\Gamma(y) = H\ \text{var}}{\lambda; \Gamma \vdash y : H\ \text{var}}\ \textsc{Var} \qquad \dfrac{\dfrac{\dfrac{\Gamma(x) = L\ \text{var}}{\lambda; \Gamma \vdash x : L\ \text{var}}\ \textsc{var}}{\lambda; \Gamma \vdash x : L}\ \textsc{r-val} \qquad L \leq H}{\lambda; \Gamma \vdash x : H}\ \textsc{SubType}}{\lambda; \{x : L\ \text{var},\ y : H\ \text{var}\} \vdash y := x : H\ \text{cmd}}\ \textsc{Assign}$$

혹시 (Subtype) 타입 규칙을 사용하면, x가 H, y가 L일 때도 그 'Bad' 프로그램에 타입을 매길 수 있을까 생각해 볼 수도 있다. 하지만 불가하다. 왜냐하면 y는 할당문의 왼편의 변수로 τ var 타입을 갖기 때문이다. 그림 3.3의 (reflex)에 의하면, τ var 타입은 자기 자신 이외의 다른 타입과 서브 타입 관계를 맺을 수 없기 때문이다.

$$\dfrac{\dfrac{\dfrac{\Gamma(y) = L\ \text{var}}{\lambda; \Gamma \vdash y : L\ \text{var}}\ \textsc{var} \qquad L\ \text{var} \not\subseteq H\ \text{var}}{\lambda; \Gamma \not\vdash y : H\ \text{var}}\ \textsc{Var} \qquad \dfrac{\Gamma(x) = H\ \text{var}}{\lambda; \Gamma \vdash x : H}\ \textsc{r\_val}}{\lambda; \{x : H\ \text{var},\ y : L\ \text{var}\} \vdash y := x : H\ \text{cmd}}\ \textsc{Assign}$$

[1] 우리의 타입 시스템에서는 격자 모델과 마찬가지로 기밀성(secrecy)과 무결성(integrity)을 동일하게 다룬다 [Den76, VIS96]. 본문에서는 기밀성 예시만 제시하지만, 동일하게 무결성에도 적용될 수 있다.

암묵적 정보 흐름 타입 매김 예제 앞의 타입 매김 규칙에서 대입문 전체에 $\tau\ cmd$라는 타입이 부여된다는 점에 주목할 필요가 있다. 이는 암묵적 흐름(implicit flow)을 통제하기 위함이다. 간단한 예를 들어 보자. 만약 x가 0 또는 1이라 하고, 다음 문장을 고려하자.

$$\texttt{if}\ \ x = 1\ \ \texttt{then}\ \ y := 1\ \ \texttt{else}\ \ y := 0$$

여기서는 x에서 y로의 명시적 흐름은 없지만, x의 값이 간접적으로 y에 복사되므로 암묵적 흐름이 발생한다. 이러한 암묵적 흐름이 안전하게 이루어지도록 하기 위해, 조건문에 대한 다음과 같은 타입 매김 규칙을 사용한다.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c : \tau\ cmd \quad \Gamma \vdash c' : \tau\ cmd}{\Gamma \vdash \texttt{if}\ \ e\ \ \texttt{then}\ \ c\ \ \texttt{else}\ \ c' : \tau\ cmd}$$

이 규칙을 직관적으로 설명하자면 다음과 같다. 조건문 분기 c와 c'는 조건식 e를 통해 보안 수준 τ의 정보가 암묵적으로 알려진 맥락에서 실행된다. 따라서 c와 c'는 τ 이상 보안 등급의 변수에만 대입할 수 있다. 이 규칙에 의하면, 조건식 e와 두 분기가 모두 동일한 보안 수준 τ를 가져야 한다. 그러나 이는 e에서 분기들로의 암묵적 상향 흐름(upward flow)을 막지는 않는다. 서브타입 매김을 통해 조건식을 상향 형변환(coercion)하거나, 혹은 명령문의 반단조성(antimonotonicity)을 이용해 분기들을 하향(coercion)하여 일치시킬 수 있다. 어떤 경우에는 두 가지 변환이 모두 필요할 수 있다. 반대로 e에서 분기들로 하향 흐름(downward flow)이 존재한다면, 일치시킬 수 없으므로 규칙은 조건문을 거부해야 한다.

예를 들어 $\Gamma(x) = \Gamma(y) = H\ var$라 하자. 앞의 대입문 규칙에 따라,

$$\Gamma \vdash y := 1 : H\ cmd, \quad \Gamma \vdash y := 0 : H\ cmd$$

를 얻을 수 있다. 이는 각 분기문이 조건식 가드(guard)를 통해 고수준 정보가 암묵적으로 알려진 맥락에서 실행될 수 있음을 의미한다. 따라서 조건문

$$\texttt{if}\ \ x = 1\ \ \texttt{then}\ \ y := 1\ \ \texttt{else}\ \ y := 0$$

은 $\tau = H$로 두었을 때

$$\Gamma \vdash \texttt{if}\ \ x = 1\ \ \texttt{then}\ \ y := 1\ \ \texttt{else}\ \ y := 0 : H\ cmd$$

로 타입이 잘 매겨진다. 이는 x와 y가 모두 높은 등급의 변수이므로, x에서 y로의 암묵적 흐름이 안전하다는 사실을 반영한다. 또한 결과 타입 $H\ cmd$는 두 분기에서 낮은 등급의 변수가 갱신되지 않았음을(no write down) 보장하며, 따라서 이 전체 조건문은 다시 고수준 정보가 암묵적으로 알려진 맥락에서 사용될 수 있다.

이제 $\Gamma(x) = L\ var$라면 암묵적 흐름은 여전히 안전하지만, 타입 시스템 내에서 이를 입증하기 위해서는 서브타입 매김이 필요하다. 한 가지 방법은 명령문의 반단조적 서브타입 관계, 즉 $L \leq H$일 때 $H\ cmd \subseteq L\ cmd$를 사용하는 것이다. 이 경우 각 분기를 $H\ cmd$에서 $L\ cmd$로 강제(coercion)하면 $\tau = L$로 두어

$$\Gamma \vdash \texttt{if}\ \ x = 1\ \ \texttt{then}\ \ y := 1\ \ \texttt{else}\ \ y := 0 : L\ cmd$$

를 얻을 수 있다.

또 다른 방법은 x의 타입을 L에서 H로 올려(coerce) $\tau = H$로 두는 것이다. 이 경우 다시 조건문의 타입은 H cmd가 된다. 이는 예컨대 이 조건문이 또 다른 조건문의 분기 안에 들어가고, 그 조건문의 가드가 높은 등급인 경우 반드시 필요한 선택이다.

마지막으로 $\Gamma(x) = H$ var이고 $\Gamma(y) = L$ var라면, 해당 조건문은 타입이 잘 매겨지지 않는다. 이는 x에서 y로의 암묵적 흐름이 하향 흐름이기 때문이며, 기대되는 동작과 일치한다.

$$\dfrac{\dfrac{\dfrac{\Gamma(x) = H \text{ var}}{\lambda; \Gamma \vdash x : H}\ \text{R\_VAR} \quad \cdots}{\lambda; \Gamma \vdash x = 1 : H}\ \textsc{Eq} \qquad \dfrac{\dfrac{\dfrac{\Gamma(y) = L \text{ var}}{\lambda; \Gamma \not\vdash y : L \text{ var}}\ \textsc{Var} \qquad L \text{ var} \not\subseteq H \text{ var}}{\lambda; \Gamma \not\vdash y : H \text{ var}}\ \textsc{Subtype} \quad \cdots}{\lambda; \Gamma \vdash y := 1 : H \text{ cmd}}\ \textsc{Assign} \quad \cdots}{\lambda; \{x : H \text{ var},\ y : L \text{ var}\} \vdash \text{if } x = 1 \text{ then } y := 1 \text{ else } y := 0 : H \text{ cmd}}\ \textsc{If}$$

Table 3.1: 조건문에 대한 세 가지 경우의 타입 판정

조건	**판정 결과**
1. $\Gamma(x) = \Gamma(y) = H$ var	$\Gamma \vdash y := 1 : H$ cmd, $\Gamma \vdash y := 0 : H$ cmd 따라서 $\Gamma \vdash$ `if` $x = 1$ `then` $y := 1$ `else` $y := 0 : H$ cmd
2. $\Gamma(x) = L$ var	암묵적 흐름은 안전하나, 서브타입 매김 H $cmd \subseteq L$ cmd를 이용해야 함. 전체 조건문의 타입은 L cmd. 또는 x의 타입을 L에서 H로 올려(coerce) $\tau = H$로 올려야 함.
3. $\Gamma(x) = H$ $var, \Gamma(y) = L$ var	하향 흐름(downward flow)이 발생하므로 조건문의 타입을 매길 수 없음

지역 변수 선언(Local Variable Declarations)

```
if x = 1 then
  letvar y := 1 in c
else
  letvar y := 0 in c'
```

Figure 3.1: 변수 x에서 변수 y로의 암묵적 흐름(implicit flow)

WHILE 언어에서 지역 변수(local variable)를 선언하는 구문이 포함되어 있다. 예를 들어, 다음과 같이 지역 변수 x를 선언한다.

$$\texttt{letvar}\ x := e\ \texttt{in}\ c$$

이는 표현식 e의 값으로 초기화된 변수 x를 생성하며, x의 유효 범위(scope)는 명령문 c이다.

초기화 과정에서 암묵적 흐름(implicit flow)이 발생할 수 있으나, 이는 항상 무해하다. 예를 들어 Figure 3.1의 코드를 살펴보자. 여기서 c와 c'는 어떤 명령문들이다. 만약 x가 high이고, 각 y가 low라면, 언뜻 보기에 x에서 y로의 하향 암묵적 흐름(downward implicit flow) 때문에 프로그램이 거부되어야 할 것처럼

보인다. 그러나 c와 c'가 어떤 low 변수도 갱신하지 않는다면(즉, 각각 high command로 타입을 매길 수 있다면), 이 프로그램은 실제로는 안전하다. 즉, x의 내용을 y를 통해 "세탁(launder)"할 수 없다.

이를 확인해 보자. 만약 x가 high라면, 앞서 제시한 조건문 타입 매김 규칙에 따라 c와 c'는 high command로 타입 매김되어야 한다. 격리 속성(confinement property)에 의해, c와 c'는 어떤 low 변수에도 대입하지 않으므로 y 또한 low 변수에 대입될 수 없다.

타입 건전성(Type Soundness)

우리의 타입 시스템에서는 두 가지 중요한 보안 성질(security lemma)을 증명할 수 있다. 바로 단순 보안(Simple Security)과 격리(Confinement)이다. Simple Security는 표현식(expression)에 적용되고, Confinement는 명령문(command)에 적용된다.

- 표현식 e가 타입 τ를 가진다면, 단순 보안 성질에 따라 기밀성(secrecy) 관점에서 e를 평가할 때 모두 τ 이하 레벨의 변수만 읽어야 한다(no read up). 무결성(integrity) 관점에서는 e에 등장하는 모든 변수가 무결성 레벨 τ의 정보를 저장해야 한다.
- 명령문 c가 타입 τ cmd를 가진다면, 격리 성질 따라 기밀성 관점에서 c 안에서는 τ 미만 레벨의 변수에 쓰지 않아야 한다(no write down). 무결성 관점에서는 c에서 갱신되는 모든 변수가 무결성 레벨 τ의 정보로부터 안전하게 갱신될 수 있어야 한다.

이 두 보안 성질은 타입 시스템의 건전성(soundness)을 증명하는 데 사용된다. 여기서 건전성은 일종의 비간섭성(noninterference) 속성으로 공식화된다. 직관적으로, 이는 타입이 잘 매겨진 프로그램에서 어떤 변수도 더 낮은 보안 수준의 변수들에 의해 간섭받지 않는다는 뜻이다. 즉, 만약 변수 v의 보안 수준이 τ라면, τ에 의해 지배되지 않는 변수들의 초기 값을 임의로 바꾸더라도, 프로그램이 정상적으로 종료되는 한, v의 최종 값은 변하지 않는다.

Table 3.2: 타입 시스템의 두 가지 보안 성질 요약

성질	기밀성(Secrecy)	무결성(Integrity)
Simple Security	표현식 e가 타입 τ를 가진다면, e 평가 시 읽는 변수는 모두 τ 이하 레벨이어야 함 (no read up)	e에 등장하는 모든 변수는 무결성 레벨 τ의 정보를 저장해야 함
Confinement	명령문 c가 타입 τ cmd를 가진다면, c 안에서는 τ 미만 레벨의 변수에 쓰지 않아야 함 (no write down)	c에서 갱신되는 모든 변수는 무결성 레벨 τ의 정보로 안전하게 갱신될 수 있어야 함

타입 추론

표준 타입 추론(type inference) 알고리즘을 사용하면, 프로그램의 타입이 잘 매겨졌는지(well typed) 여부를 자동으로 검사할 수 있다. 타입 추론에 대한 상세한 논의는 이 강의 범위를 벗어나지만, 기본적인 아이디어는 다음과 같다. 알려지지 않은 타입들을 표현하기 위해 타입 변수(type variable)를 사용하고, 프로그램이 잘 타입이 매겨지기 위해 이 타입 변수들이 만족해야 하는 제약(constraint)을 수집하는 것이다. 이러한 제약은 보통 타입 부등식(type inequality)의 형태를 가진다.

이 방법을 통해 프로그램이 가질 수 있는 모든 타입을 대표하는 대표 타입(principal type)을 구성할 수 있으며, 이는 프로그램에 부여될 수 있는 가능한 모든 타입들을 나타낸다.

3.3.3 타입 시스템의 형식적 전개 (A Formal Treatment of the Type System)

다음의 구문의 WHILE 언어를 고려한다. 이 언어는 구문 단위(phrase)들로 구성되며, 구문 단위는 표현식(expression) e 또는 명령문(command) c이다.

$$
\begin{array}{ll}
(\textit{phrases}) & p ::= e \mid c \\
(\textit{expressions}) & e ::= x \mid l \mid n \mid e + e' \mid e - e' \mid e = e' \mid e < e' \\
(\textit{commands}) & c ::= e := e' \mid c; c' \mid \texttt{if } e \texttt{ then } c \texttt{ else } c' \\
 & \quad \mid \texttt{while } e \texttt{ do } c \mid \texttt{letvar } x := e \texttt{ in } c
\end{array}
$$

메타변수 x는 식별자(identifier), l은 위치(location, 주소), n은 정수 리터럴(integer literal)을 나타낸다. 정수는 유일한 값(value)이며, 0은 false, 1은 true로 사용한다. 또한 위치들은 잘 정렬(well ordered)되어 있다고 가정한다.

이 언어에는 별도의 입출력 연산자를 두지 않았고, 모든 입출력은 프로그램 내의 자유 위치(free location)를 통해 수행한다고 가정한다. 즉, 프로그램이 입력을 읽어야 한다면 명시적 위치를 역참조(dereference)함으로써 읽으며, 출력을 해야 한다면 명시적 위치에 대입(assignment)함으로써 수행한다. 또한 프로그램 실행 중 지역 변수 선언에 의해 새로운 위치들이 생성될 수 있다. 따라서 프로그램은 입출력에 사용되는 위치뿐만 아니라 실행하면서 새롭게 위치들을 생성할 수 있다.

WHILE 언어의 타입들은 다음과 같이 구성된다.

$$
\begin{array}{ll}
(\textit{data types}) & \tau ::= s \\
(\textit{phrase types}) & \rho ::= \tau \mid \tau\ \textit{var} \mid \tau\ \textit{cmd}
\end{array}
$$

여기서 메타변수 s는 보안 클래스 집합 SC를 나타내며, SC는 부분 순서 $\leq$에 의해 정렬되어 있다고 가정한다. $\tau\ \textit{var}$는 변수 타입, $\tau\ \textit{cmd}$는 명령문 타입을 의미한다.

언어의 타입 매김 규칙(typing rule)은 Figure 3.2에 제시되어 있다. 일부 표현식의 타입 매김 규칙은 (arith) 규칙과 유사하므로 생략한다. 타입 매김 판정(typing judgment)은 다음과 같은 형태를 가진다.

$$\lambda; \Gamma \vdash p : \rho$$

여기서 λ는 위치 타입 매김(location typing), Γ는 식별자 타입 매김(identifier typing)이다. 즉, 구문 p가 타입 ρ를 가진다는 것은 λ가 p에 등장하는 위치들에 타입을 부여하고, Γ가 p의 자유 식별자들에 타입을 부여한다고 가정할 때 성립한다.

식별자 타입 매김은 식별자에서 ρ 타입으로 가는 유한 함수이며, $\Gamma(x)$는 x에 할당된 타입이다. 또한 $\Gamma\{x : \rho\}$는 x에는 타입 ρ를, 그 외의 모든 x'에는 $\Gamma(x')$를 부여하는 수정된 식별자 타입 매김이다. 위치 타입 매김(location typing)은 위치에서 τ 타입으로 가는 유한 함수이며, 표기법은 식별자 타입 매김과 유사하다.

타입 시스템의 나머지 규칙은 서브타입 매김(subtyping) 논리를 구성하며 Figure 3.3에 제시된다. 논리의 성질은 뒤에서 제시하는 보조정리(lemma)들로 증명된다.

$$(int) \quad \lambda; \Gamma \vdash n : \tau$$

$$(var) \quad \lambda; \Gamma \vdash x : \tau\ var \quad \text{if } \Gamma(x) = \tau\ var$$

$$(varloc) \quad \lambda; \Gamma \vdash l : \tau\ var \quad \text{if } \lambda(l) = \tau$$

$$(arith) \quad \frac{\lambda; \Gamma \vdash e : \tau \quad \lambda; \Gamma \vdash e' : \tau}{\lambda; \Gamma \vdash e + e' : \tau}$$

$$(r\text{-}val) \quad \frac{\lambda; \Gamma \vdash e : \tau\ var}{\lambda; \Gamma \vdash e : \tau}$$

$$(assign) \quad \frac{\lambda; \Gamma \vdash e : \tau\ var \quad \lambda; \Gamma \vdash e' : \tau}{\lambda; \Gamma \vdash e := e' : \tau\ cmd}$$

$$(compose) \quad \frac{\lambda; \Gamma \vdash c : \tau\ cmd \quad \lambda; \Gamma \vdash c' : \tau\ cmd}{\lambda; \Gamma \vdash c; c' : \tau\ cmd}$$

$$(if) \quad \frac{\lambda; \Gamma \vdash e : \tau \quad \lambda; \Gamma \vdash c : \tau\ cmd \quad \lambda; \Gamma \vdash c' : \tau\ cmd}{\lambda; \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c' : \tau\ cmd}$$

$$(while) \quad \frac{\lambda; \Gamma \vdash e : \tau \quad \lambda; \Gamma \vdash c : \tau\ cmd}{\lambda; \Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c : \tau\ cmd}$$

$$(letvar) \quad \frac{\lambda; \Gamma \vdash e : \tau \quad \lambda; \Gamma\{x : \tau\ var\} \vdash c : \tau'\ cmd}{\lambda; \Gamma \vdash \texttt{letvar}\ x := e\ \texttt{in}\ c : \tau'\ cmd}$$

Figure 3.2: 안전한 정보 흐름을 위한 WHILE 언어의 타입 매김 규칙

Lemma 3.3.1 (구조적 서브타입 매김)**.** *만약* $\vdash \rho \subseteq \rho'$*라면, 다음 중 하나가 성립한다.*

(a) ρ*는* τ *형태이고,* ρ'*는* τ' *형태이며,* $\tau \leq \tau'$*.*

(b) ρ*는* $\tau\ var$ *형태이고,* $\rho' = \rho$*.*

(c) ρ*는* $\tau\ cmd$ *형태이고,* ρ'*는* $\tau'\ cmd$ *형태이며,* $\tau' \leq \tau$*.*

Proof. $\vdash \rho \subseteq \rho'$의 유도(derivation)의 높이에 대한 귀납법으로 증명한다.

만약 유도가 규칙 (base)로 끝난다면, 규칙의 가정에 의해 (a)가 성립한다. 유도가 (reflex)로 끝난다면, $\rho = \rho'$이다. 따라서 ρ가 τ 형태라면, $\leq$가 반사적(reflexive)이므로 (a)가 성립한다. 또한 ρ가 $\tau\ var$ 형태라면 (b)가, $\tau\ cmd$ 형태라면 (c)가 각각 성립한다.

이제 유도가 (trans) 규칙으로 끝났다고 가정하자. 그러면 규칙의 가정에 의해 $\vdash \rho \subseteq \rho''$ 그리고 $\vdash \rho'' \subseteq \rho'$를 만족하는 어떤 ρ''가 존재한다. 세 가지 경우를 고려한다.

1. ρ가 τ 형태인 경우: 귀납법 가정에 의해 ρ''는 τ'' 형태이고 $\tau \leq \tau''$이다. 다시 귀납법을 적용하면 ρ'는 τ' 형태이고 $\tau'' \leq \tau'$이다. $\leq$가 추이적(transitive)이므로 $\tau \leq \tau'$가 성립한다.

2. ρ가 $\tau\ var$ 형태인 경우: 귀납법 가정에 의해 $\rho'' = \rho$이다. 다시 귀납법을 적용하면 $\rho' = \rho''$이고, 따라서 $\rho' = \rho$이다.

$$(base) \quad \frac{\tau \leq \tau'}{\vdash \tau \subseteq \tau'}$$

$$(reflex) \quad \vdash \rho \subseteq \rho$$

$$(trans) \quad \frac{\vdash \rho \subseteq \rho' \quad \vdash \rho' \subseteq \rho''}{\vdash \rho \subseteq \rho''}$$

$$(cmd\text{-}) \quad \frac{\vdash \tau \subseteq \tau'}{\vdash \tau'\ cmd \subseteq \tau\ cmd}$$

$$(subtype) \quad \frac{\lambda; \Gamma \vdash p : \rho \quad \vdash \rho \subseteq \rho'}{\lambda; \Gamma \vdash p : \rho'}$$

Figure 3.3: 서브타입 매김 규칙

3. ρ가 τ cmd 형태인 경우: 귀납법 가정에 의해 ρ''는 τ'' cmd 형태이고 $\tau'' \leq \tau$이다. 다시 귀납법을 적용하면 ρ'는 τ' cmd 형태이고 $\tau' \leq \tau''$이다. 따라서 $\leq$의 추이성에 의해 $\tau' \leq \tau$가 성립한다.

마지막으로 유도가 (cmd) 규칙으로 끝난 경우를 고려하자. 이때 ρ는 τ cmd 형태이고, ρ'는 τ' cmd 형태이며, 규칙의 가정에 의해 $\vdash \tau' \subseteq \tau$가 성립한다. 귀납법에 의해 $\tau' \leq \tau$이다. □

Lemma 3.3.2. $\subseteq$는 *부분 순서(partial order)*이다.

Proof. 반사성(reflexivity)과 추이성(transitivity)은 규칙 (reflex)과 (trans)에서 직접적으로 따라온다. 반대칭성(antisymmetry)은 Lemma 3.3.1과 $\leq$의 반대칭성으로부터 따라온다. □

3.3.4 형식적 의미론 (The Formal Semantics)

우리의 타입 시스템의 건전성(soundness)은 WHILE 언어의 프로그램(closed phrase)의 자연스러운 스타일의 의미론(natural semantics)[2]에 대해 정의할 수 있다. 구문이 닫혔다(closed)는 것은 자유 식별자(free identifier)를 갖지 않는다는 것을 의미한다.

닫힌 구문의 프로그램을 메모리 μ를 가지고 실행한다. 여기서 μ는 위치(location)에서 값(value)으로의 유한 함수이다. 위치 $l \in dom(\mu)$의 내용은 값 $\mu(l)$이며, $\mu\{l := n\}$은 위치 l에 값 n을 할당하고, $l' \neq l$인 다른 위치 l'에는 기존 값 $\mu(l')$를 유지하는 메모리를 나타낸다. 즉, $l \in dom(\mu)$인 경우 $\mu\{l := n\}$은 μ의 갱신(update)이고, 그렇지 않으면 μ의 확장(extension)이다.

실행 규칙(evaluation rule)은 Figure 3.4에 제시되어 있다. 이 규칙들은 다음과 같은 형태의 판단을 도출할 수 있도록 한다.

$$\mu \vdash e \Downarrow n \quad (\text{표현식의 경우}), \qquad \mu \vdash c \Downarrow \mu' \quad (\text{명령문의 경우})$$

즉, 닫힌 표현식 e를 메모리 μ에서 평가하면 정수 n이 되고, 닫힌 명령문 c를 메모리 μ에서 평가하면 새로운 메모리 μ'가 됨을 의미한다. 그 외의 부작용(side effect)을 일으키는 표현식은 고려하지 않으며, 명령문은 값을 결과로 내지 않는다.

[2]자연 의미론(Natural semantics)의 용어와 번역은 구조적 운영 의미론Structural Operational Semantics)과 대비되는 개념으로, Plotkin이 제안한 "big-step semantics"를 가리킨다. 프로그램 전체 실행이 한 번에 어떻게 결과 상태로 이어지는지를 정의하는 방식이라 "자연"이라는 말은 "자연스럽게 한 번에"라는 의미에서 붙였다.

기호 $[e/x]c$는 명령문 c에서 x의 모든 자유 발생을 e로 치환한, 포획 회피 치환(capture-avoiding substitution)을 의미한다. 또한 $\mu - l$은 메모리 μ에서 위치 l을 그 정의역(domain)에서 제거한 것을 의미한다.

특히 `letvar x := e in c`의 평가를 지배하는 규칙 (bindvar)에서는 치환이 사용된다. 새로운 위치 l을 c에서 x의 모든 자유 발생에 치환하여 $[l/x]c$를 얻는다. 그 결과는 확장된 메모리 $\mu[l := n]$에서 평가되며, 여기서 n은 e의 값이다. 이와 같이 치환을 사용함으로써, x를 l에 매핑하는 환경(environment)을 도입할 필요가 없다. 즉, $[l/x]c$는 부분적으로 평가된 명령문으로 볼 수 있으며, 이 안에는 여전히 다른 자유 위치들이 존재할 수 있다.

$$(base) \quad \frac{}{\mu \vdash n \Downarrow n}$$

$$(contents) \quad \frac{l \in dom(\mu)}{\mu \vdash l \Downarrow \mu(l)}$$

$$(add) \quad \frac{\mu \vdash e \Downarrow n \quad \mu \vdash e' \Downarrow n'}{\mu \vdash e + e' \Downarrow n + n'}$$

$$(update) \quad \frac{\mu \vdash e \Downarrow n \quad l \in dom(\mu)}{\mu \vdash l := e \Downarrow \mu\{l := n\}}$$

$$(sequence) \quad \frac{\mu \vdash c \Downarrow \mu' \quad \mu' \vdash c' \Downarrow \mu''}{\mu \vdash c; c' \Downarrow \mu''}$$

$$(branch) \quad \frac{\mu \vdash e \Downarrow 1 \quad \mu \vdash c \Downarrow \mu'}{\mu \vdash \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c' \Downarrow \mu'}$$

$$\frac{\mu \vdash e \Downarrow 0 \quad \mu \vdash c' \Downarrow \mu'}{\mu \vdash \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c' \Downarrow \mu'}$$

$$(loop) \quad \frac{\mu \vdash e \Downarrow 0}{\mu \vdash \texttt{while}\ e\ \texttt{do}\ c \Downarrow \mu}$$

$$\frac{\mu \vdash e \Downarrow 1 \quad \mu \vdash c \Downarrow \mu' \quad \mu' \vdash \texttt{while}\ e\ \texttt{do}\ c \Downarrow \mu''}{\mu \vdash \texttt{while}\ e\ \texttt{do}\ c \Downarrow \mu''}$$

$$(bindvar) \quad \frac{\mu \vdash e \Downarrow n \quad l \text{ is the first location not in } dom(\mu) \quad \mu\{l := n\} \vdash [l/x]c \Downarrow \mu'}{\mu \vdash \texttt{letvar}\ x := e\ \texttt{in}\ c \Downarrow \mu' - l}$$

Figure 3.4: 실행 규칙 (Evaluation Rules)

3.3.5 타입 건전성 (Type Soundness)

이제 WHILE 언어의 의미론에 대해 타입 시스템의 건전성(soundness)을 보이고자 한다. 타입 건전성 정리(the soundness theorem)는 다음을 말한다. 만약 어떤 위치 l에 대해 $\lambda(l) = \tau$라면, τ의 서브타입이 아닌 타입을 갖는 위치 l'들의 초기 값을 임의로 변경하더라도, 프로그램을 실행했을 때 (정상적으로 종료한다는 가정 하에) 최종적으로 l의 값은 변하지 않는다.

건전성 증명을 용이하게 하기 위해, 우리는 구문 구조를 따르는(syntax-directed) 타입 매김 규칙 집합을 도입한다. 이 시스템의 규칙은 Figure 3.2의 규칙에서 (r-val), (assign), (if), (while)을 제거하고, 그 대신 Figure 3.5의 구문 구조를 따르는 대응 규칙으로 교체한 것이다. Figure 3.3의 서브타입 매김 규칙은 구문 구조를 따르는 시스템에는 더이상 필요하지 않다.

$$(\textit{r-val}') \quad \frac{\lambda;\Gamma \vdash e : \tau\ var \quad \tau \leq \tau'}{\lambda;\Gamma \vdash e : \tau'}$$

$$(\textit{assign}') \quad \frac{\lambda;\Gamma \vdash e : \tau\ var \quad \lambda;\Gamma \vdash e' : \tau \quad \tau' \leq \tau}{\lambda;\Gamma \vdash e := e' : \tau'\ cmd}$$

$$(\textit{if}') \quad \frac{\lambda;\Gamma \vdash e : \tau \quad \lambda;\Gamma \vdash c : \tau\ cmd \quad \lambda;\Gamma \vdash c' : \tau\ cmd \quad \tau' \leq \tau}{\lambda;\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c' : \tau'\ cmd}$$

$$(\textit{while}') \quad \frac{\lambda;\Gamma \vdash e : \tau \quad \lambda;\Gamma \vdash c : \tau\ cmd \quad \tau' \leq \tau}{\lambda;\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c : \tau'\ cmd}$$

Figure 3.5: 구문 구조를 따르는 타입 매김 규칙 (Syntax-directed Typing Rules)

구문 구조를 따르는 시스템에서의 판단은 $\lambda;\Gamma \vdash_s p : \rho$ 형태로 쓴다. 이 시스템의 장점은, 어떤 유도에서 마지막으로 사용된 규칙이 프로그램 p와 타입 ρ의 형태에 의해 유일하게 결정된다는 점이다. 예를 들어, p가 while 루프라면, 유도는 반드시 (while′) 규칙으로 끝난다. 이는 원래 시스템에서 (while)이나 (subtype) 규칙으로 끝날 수도 있었던 것과 대조적이다.

또한 구문 구조를 따르는 규칙은 타입 추론 알고리즘(type inference algorithm)이 형변환(coercion)을 어디서 도입해야 하는지에 대한 단서를 제공한다.

다음으로, 구문 구조를 따르는 시스템이 실제로는 원래의 시스템과 동등(equivalent)함을 증명한다. 이를 위해 새로운 보조정리(lemma)가 필요하다.

Lemma 3.3.3. *만약 $\lambda;\Gamma \vdash_s p : \rho$ 이고 $\vdash \rho \subseteq \rho'$라면, $\lambda;\Gamma \vdash_s p : \rho'$이다.*

Proof. $\lambda;\Gamma \vdash_s p : \rho$의 유도 높이에 대한 귀납법으로 증명한다.

먼저, 유도가 규칙 (int)에 의해 $\lambda;\Gamma \vdash_s n : \tau$로 끝나는 경우를 보자. 이때 Lemma 3.3.1에 의해 ρ'는 τ' 형태이고, 규칙 (int)에 의해 $\lambda;\Gamma \vdash_s n : \tau'$가 성립한다.

다음으로, 유도가 규칙 (var) 또는 (varloc)에 의해 $\lambda;\Gamma \vdash_s e : \tau\ var$로 끝난다면, Lemma 3.3.1에 의해 $\rho' = \rho$이다.

유도가 규칙 (arith)에 의해 $\lambda;\Gamma \vdash_s e+e' : \tau$로 끝나는 경우를 보자. 이때 $\lambda;\Gamma \vdash_s e : \tau$ 그리고 $\lambda;\Gamma \vdash_s e' : \tau$가 성립한다. Lemma 3.3.1에 의해 ρ'는 τ' 형태이다. 귀납법 가정에 따라 $\lambda;\Gamma \vdash_s e : \tau'$ 및 $\lambda;\Gamma \vdash_s e' : \tau'$가 성립한다. 따라서 규칙 (arith)에 의해 $\lambda;\Gamma \vdash_s e + e' : \tau'$가 성립한다. 유도가 규칙 (compose)이나 (letvar)로 끝나는 경우도 이와 유사하다.

이제 유도가 규칙 (r-val′)에 의해 $\lambda;\Gamma \vdash_s e : \tau$로 끝나는 경우를 보자. 이때 어떤 타입 τ''가 존재하여 $\lambda;\Gamma \vdash_s e : \tau''\ var$이고 $\tau'' \leq \tau$이다. Lemma 3.3.1에 의해 ρ'는 τ' 형태이고 $\tau \leq \tau'$이다. $\leq$가 추이적이므로 $\tau'' \leq \tau'$이고, 따라서 규칙 (r-val′)에 의해 $\lambda;\Gamma \vdash_s e : \tau'$가 성립한다.

유도가 규칙 (assign′)에 의해 $\lambda;\Gamma \vdash_s e := e' : \tau\ cmd$로 끝나는 경우를 보자. 이때 어떤 타입 τ''가 존재하여 $\lambda;\Gamma \vdash_s e : \tau''\ var$, $\lambda;\Gamma \vdash_s e' : \tau''$, 그리고 $\tau \leq \tau''$가 성립한다. Lemma 3.3.1에 의해 ρ'는 $\tau'\ cmd$ 형태이고 $\tau' \leq \tau$이다. $\leq$가 추이적이므로 $\tau' \leq \tau''$이고, 따라서 규칙 (assign′)에 의해 $\lambda;\Gamma \vdash_s e := e' : \tau'\ cmd$가 성립한다.

유도가 규칙 (if′)나 (while′)로 끝나는 경우도 이와 유사하게 다룰 수 있다. □

이제 동치성은 다음 정리로 표현된다.

Theorem 3.3.4. $\lambda;\Gamma \vdash p : \rho$ *iff* $\lambda;\Gamma \vdash_s p : \rho$.

Proof. $\boxed{\lambda;\Gamma \vdash_s p : \rho \;\Rightarrow\; \lambda;\Gamma \vdash p : \rho}$: 먼저 $\lambda;\Gamma \vdash_s p : \rho$라 하자. 이 경우 $\lambda;\Gamma \vdash p : \rho$임을 쉽게 확인할 수 있다. 왜냐하면 구문 구조를 따르는 규칙 (r-val′), (assign′), (if′), (while′)의 각 사용은 규칙 (r-val), (assign), (if), (while)의 사용 뒤에 (subtype)을 적용함으로써 흉내낼 수 있기 때문이다.

예를 들어, (assign′) 규칙의 사용

$$\frac{\lambda;\Gamma \vdash e : \tau\ var \quad \lambda;\Gamma \vdash e' : \tau \quad \tau' \leq \tau}{\lambda;\Gamma \vdash e := e' : \tau'\ cmd}$$

은 다음과 같이 흉내낼 수 있다. 먼저 (assign)을 사용하여 $\lambda;\Gamma \vdash e := e' : \tau\ cmd$를 보이고, (base)와 (cmd)를 사용하여 $\vdash \tau\ cmd \subseteq \tau'\ cmd$를 보인 다음, (subtype)을 사용하여 $\lambda;\Gamma \vdash e := e' : \tau'\ cmd$를 보일 수 있다.

$\boxed{\lambda;\Gamma \vdash p : \rho \;\Rightarrow\; \lambda;\Gamma \vdash_s p : \rho}$: 이제 $\lambda;\Gamma \vdash p : \rho$라고 가정하자. 귀납법(induction)을 통해 $\lambda;\Gamma \vdash_s p : \rho$임을 보일 것이다. 귀납은 $\lambda;\Gamma \vdash p : \rho$의 유도 높이에 대해 진행한다.

- 만약 유도가 (int), (var), (varloc)로 끝난다면, $\lambda;\Gamma \vdash_s p : \rho$는 즉시 성립한다.
- 유도가 (arith), (compose), (letvar)로 끝난 경우에도 귀납 가정에 의해 직접적으로 성립한다.
- 유도가 (r-val), (assign), (if), (while)로 끝난 경우에는 $\leq$가 반사적(reflexive)이라는 사실을 이용하여, 해당하는 구문 구조를 따르는 규칙을 적용함으로써 $\lambda;\Gamma \vdash_s p : \rho$를 얻는다.

마지막으로, 유도가 (subtype)으로 끝나는 경우를 보자. 이때 규칙의 가정에 의해 어떤 타입 ρ'가 존재하여 $\lambda;\Gamma \vdash p : \rho'$ 그리고 $\vdash \rho' \subseteq \rho$가 성립한다. 귀납법 가정에 의해 $\lambda;\Gamma \vdash_s p : \rho'$이고, 따라서 Lemma 3.3.3에 의해 $\lambda;\Gamma \vdash_s p : \rho$가 성립한다. □

앞으로는 모든 타이핑 유도가 구문 구조를 따르는 타입 시스템에서 이루어진다고 가정한다. 따라서 $\vdash$는 $\vdash_s$를 의미한다.

최종 준비로, 타입 시스템과 의미론의 다음 성질들을 보인다.

Lemma 3.3.5 (간단 보안(Simple Security))**.** *만약* $\lambda;\Gamma \vdash e : \tau$*라면,* e*에 등장하는 모든 위치* l*에 대해* $\lambda(l) \leq \tau$*이다.*

Proof. e의 구조에 대한 귀납법으로 증명한다.

먼저, 규칙 (r-val′)에 의해 $\lambda;\Gamma \vdash l : \tau$라고 가정하자. 그러면 어떤 타입 τ'가 존재하여 $\lambda;\Gamma \vdash l : \tau'\ var$이고 $\tau' \leq \tau$이다. 규칙 (varloc)에 의해 $\lambda(l) = \tau'$이므로, $\lambda(l) \leq \tau$가 성립한다.

다음으로, $\lambda;\Gamma \vdash e + e' : \tau$라고 가정하자. 그러면 $\lambda;\Gamma \vdash e : \tau$ 그리고 $\lambda;\Gamma \vdash e' : \tau$가 성립한다. 귀납법을 두 번 적용하면, e에 속하는 모든 위치 l에 대해 $\lambda(l) \leq \tau$, 그리고 e'에 속하는 모든 위치 l에 대해서도 $\lambda(l) \leq \tau$가 성립한다. 따라서 $e + e'$에 속하는 모든 위치 l에 대해 $\lambda(l) \leq \tau$임을 알 수 있다. □

단순 보안(Simple Security)은 기밀성과 무결성 모두에 적용된다.

- 기밀성(secrecy)의 경우: 표현식 e가 평가될 때 τ 이하 등급의 위치들만 읽힐 수 있다 (**no read up**). 예를 들어, $L \leq H$이고 $\tau = L$이라면, e는 어떤 H 위치도 읽지 않고 평가될 수 있다.
- 무결성(integrity)의 경우: e의 무결성 수준이 τ라면, e에 등장하는 모든 위치는 무결성 수준 τ의 정보를 저장해야 한다. 예를 들어, $T \leq U$ (여기서 T는 trusted, U는 untrusted)이고 $\tau = T$라면, e에 등장하는 모든 위치는 신뢰된 정보를 저장한다.

Lemma 3.3.6 (격리(Confinement))**.** *만약 $\lambda; \Gamma \vdash c : \tau\ cmd$라면, c에서 대입되는 모든 위치 l에 대해 $\lambda(l) \geq \tau$이다.*

Proof. c의 구조에 대한 귀납법으로 증명한다.

먼저, 규칙 (assign′)에 의해 $\lambda; \Gamma \vdash l := e : \tau\ cmd$라고 가정하자. 그러면 어떤 타입 τ'가 존재하여 $\lambda; \Gamma \vdash l : \tau'\ var$, $\lambda; \Gamma \vdash e : \tau'$, 그리고 $\tau \leq \tau'$가 성립한다. 규칙 (varloc)에 의해 $\lambda(l) = \tau'$이므로, $\lambda(l) \geq \tau$가 성립한다.

c가 두 명령문의 합성(composition) 혹은 `letvar` 명령문인 경우, 귀납법에 의해 정리가 직접적으로 성립한다.

다음으로, 규칙 (while′)에 의해 $\lambda; \Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c' : \tau\ cmd$라고 가정하자. 그러면 어떤 타입 τ'가 존재하여 $\lambda; \Gamma \vdash e : \tau'$, $\lambda; \Gamma \vdash c' : \tau'\ cmd$, 그리고 $\tau \leq \tau'$가 성립한다. 귀납법에 의해 c'에서 대입되는 모든 위치 l에 대해 $\lambda(l) \geq \tau'$이다. $\geq$가 추이적이므로, c'에서 대입되는 모든 위치 l에 대해 $\lambda(l) \geq \tau$가 성립한다. 따라서 $\texttt{while}\ e\ \texttt{do}\ c'$에서 대입되는 모든 위치 l에 대해서도 $\lambda(l) \geq \tau$이다.

c가 조건문(conditional)인 경우도 이와 유사하게 다룰 수 있다. □

격리(Confinement) 또한 기밀성과 무결성 모두에 적용된다.

- 기밀성의 경우: c 안에서는 τ 미만 레벨의 위치는 갱신되지 않는다 (no write down).
- 무결성의 경우: c에서 대입되는 모든 위치는 무결성 수준 τ의 정보로 안전하게 갱신될 수 있어야 한다. 예를 들어, $\tau = U$라면, 이 보조정리는 c가 평가될 때 어떤 신뢰된(trusted) 위치도 갱신되지 않음을 보장한다.

다음 보조정리는 [Har94]에 제시된 보조정리의 단순한 변형이다.

Lemma 3.3.7 (대체(Substitution))**.** *만약 $\lambda; \Gamma \vdash l : \tau\ var$이고 $\lambda; \Gamma[x : \tau\ var] \vdash c : \tau'\ cmd$라면, $\lambda; \Gamma \vdash [l/x]c : \tau'\ cmd$이다.*

Lemma 3.3.8. *만약 $\mu \vdash c \Downarrow \mu'$라면, $\mathrm{dom}(\mu) = \mathrm{dom}(\mu')$이다.*

Lemma 3.3.9. *만약 $\mu \vdash c \Downarrow \mu'$, $l \in dom(\mu)$, 그리고 c에서 l이 대입되지 않는다면, $\mu(l) = \mu'(l)$이다.*

Lemma 3.3.8과 3.3.9, 모두 $\mu \vdash c \Downarrow \mu'$의 유도 구조에 대한 귀납법으로 쉽게 증명할 수 있다.

이제 타입 건전성 정리를 증명할 준비가 되었다.

Theorem 3.3.10 (타입 건전성(Type Soundness))**.** *다음이 성립한다고 가정하자.*

(a) $\lambda \vdash c : \rho$,

(b) $\mu \vdash c \Downarrow \mu'$,

(c) $\nu \vdash c \Downarrow \nu'$,

(d) $dom(\mu) = dom(\nu) = dom(\lambda)$,

(e) $\nu(l) = \mu(l)$ *for all* l *such that* $\lambda(l) \leq \tau$.

그렇다면 $\nu'(l) = \mu'(l)$ *for all* l *such that* $\lambda(l) \leq \tau$.

Proof. $\mu \vdash c \Downarrow \mu'$의 유도 구조에 대한 귀납법으로 증명한다. 여기서는 세 가지 경우만 다룬다: (update), (loop), (bindvar). 나머지 평가 규칙들은 유사하게 처리할 수 있다.

(update) μ에서의 평가가 다음과 같이 끝난다고 가정하자:

$$\frac{\mu \vdash e \Downarrow n \quad l \in dom(\mu)}{\mu \vdash l := e \Downarrow \mu[l := n]}$$

ν에서의 평가가

$$\frac{\nu \vdash e \Downarrow n' \quad l \in dom(\nu)}{\nu \vdash l := e \Downarrow \nu[l := n']}$$

로 끝나며, 타입 유도는 (assign′) 규칙에 의해

$$\frac{\lambda \vdash l : \tau_2\ var \quad \lambda \vdash e : \tau_2 \quad \tau_1 \leq \tau_2}{\lambda \vdash l := e : \tau_1\ cmd}$$

으로 끝난다고 하자. 두 가지 경우가 있다.

1. $\tau_2 \leq \tau$인 경우: Simple Security Lemma에 의해 e에 속한 모든 l'에 대해 $\lambda(l') \leq \tau_2$이다. $\leq$의 추이성에 의해 e의 모든 l'에 대해 $\lambda(l') \leq \tau$가 된다. 따라서 가정 (e)에 의해 e의 모든 l'에 대해 $\mu(l') = \nu(l')$, 따라서 $n = n'$이다. 그러므로 $\lambda(l') \leq \tau$인 모든 l'에 대해 $\mu\{l := n\}(l') = \nu\{l := n'\}(l')$가 성립한다.

2. $\tau_2 \not\leq \tau$인 경우: 규칙 (varloc)에 의해 $\lambda(l) = \tau_2$이고, 따라서 $\lambda(l) \not\leq \tau$이다. 그러므로 가정 (e)에 의해 $\lambda(l') \leq \tau$인 모든 l'에 대해 $\mu\{l := n\}(l') = \nu\{l := n'\}(l')$가 성립한다.

(loop) $\mu \vdash \texttt{while}\ e\ \texttt{do}\ c \Downarrow \mu'$, $\nu \vdash \texttt{while}\ e\ \texttt{do}\ c \Downarrow \nu'$이고, 타입 유도가 (while′) 규칙에 의해

$$\frac{\lambda \vdash e : \tau_2 \quad \lambda \vdash c : \tau_2\ cmd \quad \tau_1 \leq \tau_2}{\lambda \vdash \texttt{while}\ e\ \texttt{do}\ c : \tau_1\ cmd}$$

로 끝난다고 하자. 두 가지 경우가 있다.

1. $\tau_2 \leq \tau$인 경우: Simple Security Lemma에 의해 e에 속한 모든 l에 대해 $\lambda(l) \leq \tau_2$, 따라서 $\lambda(l) \leq \tau$이다. 따라서 가정 (e)에 의해 e의 모든 l에 대해 $\mu(l) = \nu(l)$이고, 따라서 $\mu \vdash e \Downarrow n$, $\nu \vdash e \Downarrow n$이다. - 만약 $e \Downarrow 0$이라면, $\mu \vdash \texttt{while}\ e\ \texttt{do}\ c \Downarrow \mu$ 그리고 $\nu \vdash \texttt{while}\ e\ \texttt{do}\ c \Downarrow \nu$이다. 이때 $\lambda(l) \leq \tau$인 모든 l에 대해 $\mu(l) = \nu(l)$이므로 증명이 끝난다. - 만약 $e \Downarrow 1$이라면,

$$\mu \vdash e \Downarrow 1, \quad \mu \vdash c \Downarrow \mu_1, \quad \mu_1 \vdash \texttt{while}\ e\ \texttt{do}\ c \Downarrow \mu_2$$

$$\nu \vdash e \Downarrow 1, \quad \nu \vdash c \Downarrow \nu_1, \quad \nu_1 \vdash \texttt{while}\ e\ \texttt{do}\ c \Downarrow \nu_2$$

귀납법 가정에 의해 $\lambda(l) \leq \tau$인 모든 l에 대해 $\mu_1(l) = \nu_1(l)$. Lemma 3.3.8에 의해 $dom(\mu) = dom(\mu_1)$, $dom(\nu) = dom(\nu_1)$. 가정 (d)에 의해 $dom(\mu_1) = dom(\nu_1) = dom(\lambda)$. 다시 귀납법을 적용하면 $\lambda(l) \leq \tau$인 모든 l에 대해 $\mu_2(l) = \nu_2(l)$.

2. $\tau_2 \not\leq \tau$인 경우: Confinement Lemma에 의해 c에서 대입되는 모든 l에 대해 $\lambda(l) \geq \tau_2$. 따라서 $\lambda(l) \leq \tau$라면 $\tau_2 \leq \tau$이어야 하는데, 이는 모순이다. 따라서 $\lambda(l) \leq \tau$인 l은 c나 $\texttt{while}\ e\ \texttt{do}\ c$에서 대입되지 않는다. Lemma 3.3.7에 의해 $\lambda(l) \leq \tau$인 모든 l에 대해 $\mu'(l) = \mu(l)$, $\nu'(l) = \nu(l)$. 따라서 가정 (e)에 의해 $\mu'(l) = \nu'(l)$.

(bindvar) μ에서의 평가가

$$\frac{\mu \vdash e \Downarrow n \quad l \notin dom(\mu) \quad \mu\{l := n\} \vdash [l/x]c \Downarrow \mu'}{\mu \vdash \texttt{letvar}\ x := e\ \texttt{in}\ c \Downarrow \mu' - l}$$

로 끝나고, $dom(\mu) = dom(\nu)$이므로 ν에서의 평가가

$$\frac{\nu \vdash e \Downarrow n' \quad l \notin dom(\nu) \quad \nu\{l := n'\} \vdash [l/x]c \Downarrow \nu'}{\nu \vdash \texttt{letvar}\ x := e\ \texttt{in}\ c \Downarrow \nu' - l}$$

로 끝나며, 타입 유도는 (letvar) 규칙에 의해

$$\frac{\lambda \vdash e : \tau_1 \quad \lambda; \{x : \tau_1\ var\} \vdash c : \tau_2\ cmd}{\lambda \vdash \texttt{letvar}\ x := e\ \texttt{in}\ c : \tau_2\ cmd}$$

로 끝난다고 하자.

규칙 (varloc)에 의해 $\lambda\{l : \tau_1\} \vdash l : \tau_1\ var$이다. 가정 (d)와 $l \notin dom(\mu)$로부터 $l \notin dom(\lambda)$이다. 따라서 $\lambda\{l : \tau_1\}; \{x : \tau_1\ var\} \vdash c : \tau_2\ cmd$. Lemma 3.3.7에 의해 $\lambda[l : \tau_1] \vdash [l/x]c : \tau_2\ cmd$. 또한 $dom(\mu\{l := n\}) = dom(\nu\{l := n'\}) = dom(\lambda\{l : \tau_1\})$이다.

귀납법을 적용하려면, 모든 $\lambda\{l : \tau_1\}(l') \leq \tau$인 l'에 대해 $\nu\{l := n'\}(l') = \mu\{l := n\}(l')$임을 보이면 충분하다.

- 만약 $l' \neq l$이면, 가정 (e)에 의해 성립한다.
- 만약 $l' = l$이라면, $\tau_1 \leq \tau$일 경우 $n = n'$임을 보여야 한다. Simple Security Lemma에 의해 e의 모든 l''에 대해 $\lambda(l'') \leq \tau_1$이다. 따라서 $\tau_1 \leq \tau$라면 e의 모든 l''에 대해 $\lambda(l'') \leq \tau$이다. 가정 (e)에 의해 e의 모든 l''에 대해 $\mu(l'') = \nu(l'')$, 따라서 $n = n'$이다.

따라서 귀납법에 의해 $\lambda[l : \tau_1](l'') \leq \tau$인 모든 l''에 대해 $\nu'(l'') = \mu'(l'')$. 결국 $\lambda(l'') \leq \tau$인 모든 l''에 대해 $\nu' - l(l'') = \mu' - l(l'')$가 성립한다. □

3.4 연습문제

다음 프로그램의 정보 흐름을 분석해보자.

$$\texttt{if } x > y \texttt{ then } z := w \texttt{ else } i := i + 1$$

위 코드에서는 조건식에 사용되는 변수 x, y로부터 대입문에 포함된 변수 z, i로 **암묵적 정보 흐름(implicit flow)** 이 발생한다. 정보 흐름 보안(type system 기반 비밀성 보장)을 가정할 때, 변수 i, x, y, w, z 각각에 대해 부여 가능한 보안 수준(예: L, H)의 제약 조건을 타입 규칙을 통해 분석하시오.

두 점 격자 $L \sqsubseteq H$ 에 대해, 가능한 security level 배치와 정보 흐름이 안전한지 여부에 대한 참고 예시이다.

안전한 예시

- $x = L,\ y = L,\ w = L,\ z = L,\ i = L$
- $x = L,\ y = L,\ w = H,\ z = H,\ i = L$
- $x = H,\ y = L,\ w = L,\ z = H,\ i = H$
- $x = H,\ y = H,\ w = H,\ z = H,\ i = H$

불안전한 예시

- $x = H$ (또는 $y = H$) 인데 $i = L$
- $x = H$ (또는 $y = H$) 인데 $z = L$
- $x = y = L,\ w = H,\ z = L$

Chapter 4

정적 분석: 자료 흐름 분석

live: of continuing or current interest

— *Webster's Dictionary*

컴파일러의 프런트엔드는 프로그램을 무한 개의 임시 변수(temporaries)를 갖는 중간 언어(intermediate language)로 변환한다. 그러나 이 프로그램은 한정된 개수의 레지스터만 가진 머신에서 실행되어야 한다. 두 개의 임시 변수 a와 b가 동시에 사용되지 않는다면, 하나의 레지스터를 함께 사용할 수 있다. 즉, 많은 임시 변수들이 소수의 레지스터에 들어갈 수 있다. 만약 모두 수용되지 않더라도, 초과되는 임시 변수는 메모리에 보관하면 된다.

따라서 컴파일러는 중간 표현 프로그램을 분석하여 어떤 임시 변수들이 동시에 사용 중인지 알아내야 한다. 어떤 변수의 값이 앞으로 필요할 가능성이 있다면 그 변수를 *라이브(live)* 상태라고 하며, 이러한 분석을 *라이브니스 분석(liveness analysis)*이라고 한다.

프로그램을 분석할 때는 *제어 흐름 그래프(control-flow graph)*를 만드는 것이 유용한 경우가 많다. 프로그램의 각 문장은 흐름 그래프의 하나의 노드가 된다. 만약 문장 x 다음에 문장 y가 실행될 수 있다면, x에서 y로 향하는 간선(edge)이 있는 것이다. 그림 4.1은 간단한 루프에 대한 흐름 그래프를 보여준다.

이제 각 변수의 라이브니스를 살펴보자(그림 4.2). 변수의 현재 값이 미래에 사용될 예정이라면 그 변수는 라이브 상태라고 본다. 따라서 라이브니스 분석은 보통 미래에서 과거 방향으로 분석을 진행한다. 변수 b는 문장 4에서 사용되므로, b는 $3 \to 4$ 간선에서 라이브이다. 문장 3은 b에 값을 대입하지 않기 때문에 b는 $2 \to 3$ 간선에서도 라이브 상태이다. 반면, 문장 2는 b에 값을 대입한다. 따라서 $1 \to 2$ 간선에 존재하는 b의 내용은 더 이상 누구에게도 필요하지 않게 된다. 즉, 이 구간에서 b는 데드(dead) 상태이다. 따라서 b의 라이브 구간(live range)은 $\{2 \to 3,\ 3 \to 4\}$ 이다.

변수 a는 흥미로운 경우이다. a는 $1 \to 2$ 구간에서 라이브이며, 다시 $4 \to 5 \to 2$ 경로에서도 라이브 상태가 된다. 그러나 $2 \to 3 \to 4$ 구간에서는 라이브하지 않다. 비록 a가 완벽히 노드 3에서 잘 정의된 값을 가지고 있더라도, 그 값은 a에 새로운 값이 대입되기 전까지는 다시 사용되지 않는다.

변수 c는 이 프로그램에 진입하는 시점에서 이미 라이브 상태이다. 아마도 그것은 형식 매개변수(formal parameter)일 수 있다. 만약 c가 지역 변수(local variable)라면, 라이브니스 분석은 초기화되지 않은 변수를 감지한 것이다; 이 경우 컴파일러는 프로그래머에게 경고 메시지를 출력할 수도 있다.

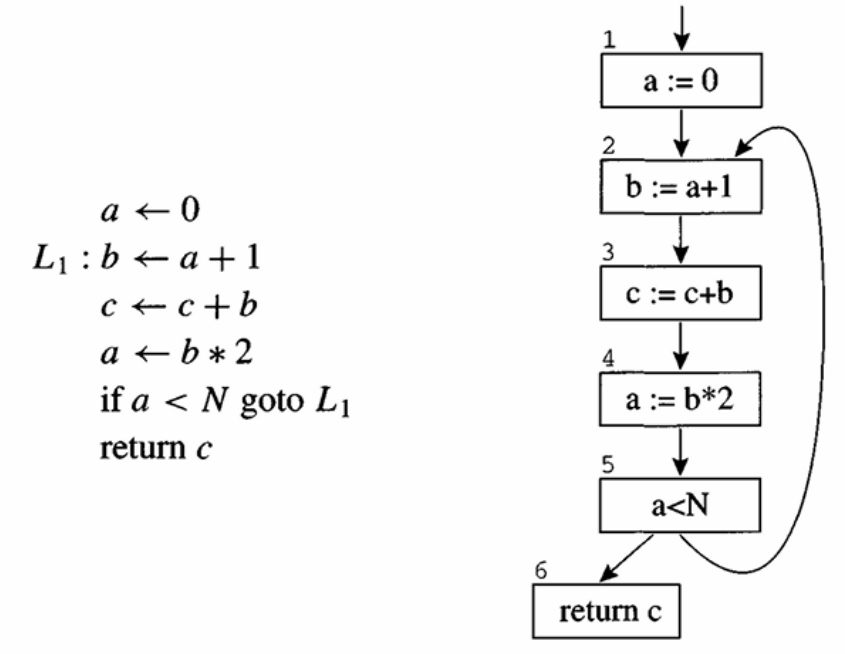


Figure 4.1: 프로그램의 제어 흐름 그래프

모든 변수의 라이브 구간이 계산되고 나면, a, b, c를 저장하기 위해 두 개의 레지스터만 필요하다는 것을 알 수 있다. 왜냐하면 a와 b는 동시에 라이브 상태가 되는 일이 없기 때문이다. 레지스터 1은 a와 b를 모두 저장할 수 있고, 레지스터 2는 c를 저장하면 된다.

4.1 데이터플로 방정식의 해법

변수의 라이브니스(liveness)는 제어 흐름 그래프(control-flow graph)의 간선을 따라 "흐른다"; 각 변수의 라이브 구간(live range)을 결정하는 것은 데이터플로(dataflow) 문제의 한 예이다. 17장에서는 이와 다른 여러 종류의 데이터플로 문제를 다룰 것이다.

흐름 그래프 용어(Flow graph terminology). 흐름 그래프의 한 노드는 후속 노드(successor nodes)로 이어지는 *out-edge*들을 가지고 있으며, 선행 노드(predecessor nodes)로부터 들어오는 *in-edge*들을 가진다. 집합 pred[n]은 노드 n의 모든 선행 노드들이며, succ[n]은 후속 노드들의 집합이다.

그래프 4.1에서 노드 5의 out-edge는 $5 \to 6$과 $5 \to 2$이고, 따라서 succ[5] $= \{2, 6\}$이다. 노드 2의 in-edge는 $5 \to 2$와 $1 \to 2$이므로, pred[2] $= \{1, 5\}$이다.

사용과 정의(Uses and defs). 변수 또는 임시 변수에 대한 대입(assign)은 그 변수를 *정의(defines)*한다. 대입문의 오른쪽(또는 다른 표현식)에서 변수가 등장하면, 그 변수는 *사용(uses)*된 것이다. 우리는 변수의 *def*를 그것을 정의하는 그래프 노드들의 집합으로 정의할 수 있으며, 반대로 특정 그래프 노드의 *def*는 그 노드가 정의하는 변수들의 집합으로 정의할 수 있다. 마찬가지로 변수 또는 그래프 노드의 *use* 역시 정의할 수 있다.

그래프 4.1에서 def(3) $= \{c\}$이고, use(3) $= \{b, c\}$이다.

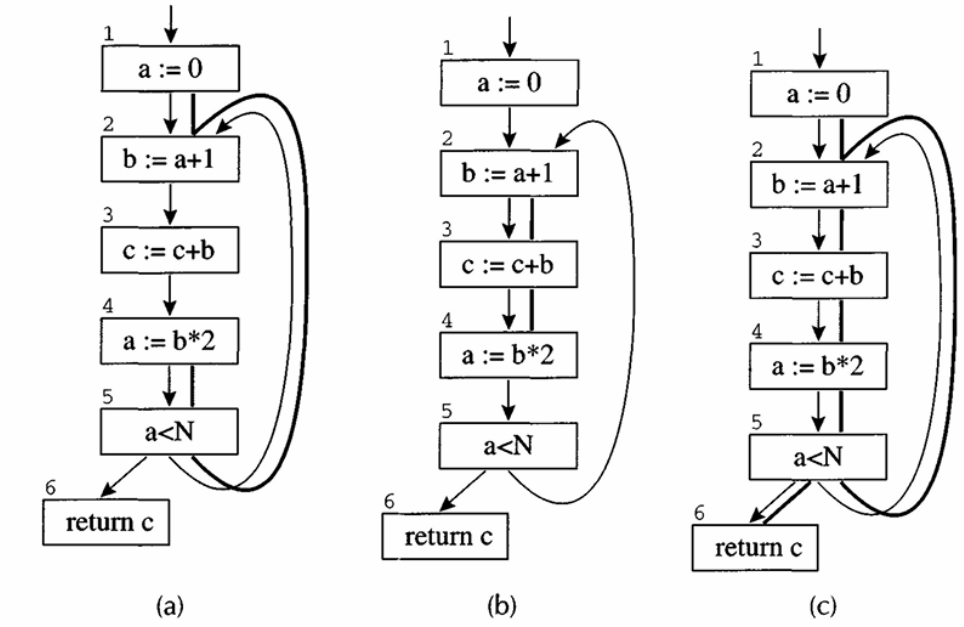


Figure 4.2: Liveness of variables a, b, c.

라이브니스(Liveness). 변수는 어떤 간선(edge) 위에서 *라이브(live)* 상태일 수 있는데, 그 간선에서 시작하여 그 변수가 정의(def)되지 않은 채 사용(use)되는 경로가 존재하는 경우를 말한다. 변수가 어떤 노드에서 *live-in* 상태라는 것은, 그 노드의 in-edge 중 하나에서 라이브 상태임을 의미한다. 반대로, *live-out* 상태라는 것은 그 노드의 out-edge 중 하나에서 라이브 상태임을 의미한다.

라이브니스 계산(Calculation of Liveness). 라이브니스 정보(*live-in* 및 *live-out*)는 *use*와 *def* 집합을 기반으로 다음과 같이 계산할 수 있다.

1. 만약 어떤 변수가 use[n]에 포함된다면, 그 변수는 노드 n에서 *live-in* 상태이다. 즉, 어떤 문장이 변수를 사용한다면, 그 변수는 그 문장에 진입하는 시점에서 라이브 상태라는 뜻이다.

2. 만약 어떤 변수가 노드 n에서 *live-in* 상태라면, 그 변수는 pred[n]의 모든 노드 m에서 *live-out* 상태이다.

3. 만약 어떤 변수가 노드 n에서 *live-out* 상태이며, 동시에 def[n]에 포함되지 않는다면, 그 변수는 노드 n에서도 *live-in* 상태이다. 즉, 누군가가 문장 n의 끝에서 변수 a의 값을 필요로 하지만, 문장 n이 그 값을 제공하지 않는다면, 그 값은 문장 n의 진입 시점에서도 필요하다는 뜻이다.

이 세 문장은 변수의 집합에 대한 방정식 4.3으로 표현될 수 있다. *live-in* 집합은 노드별로 인덱싱된 배열 in[n]이고, *live-out* 집합은 out[n]이다. 즉, in[n]은 use[n]에 있는 모든 변수들과, out[n]에 있으면서 def[n]에는 없는 변수들의 합집합이다. 반면, out[n]은 노드 n의 모든 후속 노드에 대한 in 집합의 합집합이다.

알고리즘 4.4는 이 방정식들을 반복(iteration)으로 해결한다. 일반적으로 우리는 모든 n에 대해 in[n]과 out[n]을 빈 집합 {}으로 초기화한 후, 고정점(fixed point)에 도달할 때까지 이 방정식들을 대입문처럼 반복해서 적용한다.

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \qquad \text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

Figure 4.3: Dataflow equations for liveness analysis.

for each n
  $\text{in}[n] \leftarrow \{\}$; $\text{out}[n] \leftarrow \{\}$
repeat
  for each n
    $\text{in}'[n] \leftarrow \text{in}[n]$; $\text{out}'[n] \leftarrow \text{out}[n]$
    $\text{in}[n] \leftarrow \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$
    $\text{out}[n] \leftarrow \bigcup_{s \in \text{succ}[n]} \text{in}[s]$
until $\text{in}'[n] = \text{in}[n]$ **and** $\text{out}'[n] = \text{out}[n]$ **for all** n

Figure 4.4: Computation of liveness by iteration.

표 4.5는 그래프 4.1에 대해 알고리즘을 실행한 결과를 보여준다. 각 열(1st, 2nd 등)은 반복문(`repeat` 루프)의 각 차수(iteration)에서의 in과 out 값들을 나타낸다. 7번째 열이 6번째 열과 동일하므로, 알고리즘은 수렴했다고 볼 수 있다.

			1st		2nd		3rd		4th		5th		6th		7th	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c		c	

Figure 4.5: Liveness calculation following forward control-flow edges.

이 알고리즘의 수렴 속도를 높이기 위해 노드를 적절한 순서로 처리할 수 있다. 예를 들어 그래프에서 간선 3 → 4가 있을 때, in[4]는 out[4]로부터 계산되며, out[3]은 in[4]로부터 계산된다. 따라서 우리는 out[4] → in[4] → out[3] → in[3] 순서로 in과 out을 계산해야 한다. 하지만 표 4.5에서는 반복마다 정확히 반대 순서가 사용되고 있다. 우리는 이전 반복에서 얻은 정보를 최대한 활용하기 위해 (각 반복마다) 가능한 늦게 값을 계산한 셈이다.

표 4.6은 각 `for` 반복이 6에서 1까지 진행되는 방식으로 계산된 결과를 보여준다 (이는 흐름 그래프 화살표의 *역방향*을 따르는 것과 유사하다). 그리고 각 반복에서 out 집합이 in 집합보다 먼저 계산된다. 두 번째 반복이 끝날 때 고정점이 발견되며, 세 번째 반복은 이를 다시 확인할 뿐이다.

데이터플로 방정식을 반복(iteration)으로 해결할 때는 계산 순서를 "흐름(flow)"에 맞춰야 한다. 라이브니스는 제어 흐름 그래프의 화살표를 따라 *역방향*으로 흐르므로, "out에서 in으로" 계산 순서도 그에 맞춰야 한다.

노드의 순서를 정하는 것은 17.4절에서 설명된 바와 같이 깊이 우선 탐색(depth-first search)으로 쉽게 수행할 수 있다.

			1st		2nd		3rd	
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

Figure 4.6: Liveness calculation following reverse control-flow edges.

기본 블록(Basic blocks). 선행 노드와 후속 노드가 각각 하나씩만 있는 흐름 그래프의 노드들은 큰 의미가 없다. 이러한 노드들은 그들의 선행/후속 노드와 병합될 수 있으며, 그 결과 그래프는 훨씬 더 적은 수의 노드로 구성되고, 각 노드는 하나의 기본 블록(basic block)을 나타내게 된다. 라이브니스 분석과 같은 흐름 그래프 기반의 알고리즘은 이러한 더 작은 그래프에서 훨씬 더 빠르게 동작한다. 17장에서는 데이터플로 방정식을 기본 블록에 맞게 조정하는 방법을 설명할 것이며, 본 장에서는 단순한 형태로 유지한다.

한 번에 하나의 변수(One variable at a time). 집합 방정식을 이용하여 데이터플로를 "병렬(parallel)" 방식으로 계산하는 대신, 필요할 때마다 하나의 변수에 대해 데이터플로를 계산하는 것도 실용적인 방법이다. 라이브니스 분석의 경우, 이는 각 임시 변수에 대해 데이터플로 순회를 반복한다는 의미이다. 즉, 임시 변수 t가 사용되는(use) 모든 지점에서 시작하여, 흐름 그래프의 선행 간선을 따라 깊이 우선 탐색(depth-first search) 방식으로 역방향 추적하며, 각 노드에서 t가 라이브 상태임을 기록한다. 탐색은 해당 임시 변수가 정의되는 지점에서 멈춘다. 이는 다소 비용이 들어 보일 수 있지만, 많은 임시 변수들은 라이브 구간이 매우 짧기 때문에 대부분의 경우 탐색은 빠르게 종료되며, 흐름 그래프 전체를 모두 순회하지 않는다.

집합의 표현(Representation of sets). 데이터플로 방정식에서 사용할 집합을 표현하는 방법은 크게 두 가지가 있다: 비트 배열(bit array)로 표현하거나, 정렬된 리스트(sorted list)로 표현하는 방법이 있다.

프로그램에 N개의 변수가 있다면, 비트 배열 방식은 각 집합을 위해 N개의 비트를 사용한다. 두 집합의 합집합(union)을 계산하는 것은 해당 위치의 비트를 OR 연산으로 계산하면 된다. 일반적인 컴퓨터는 하나의 워드(word)당 K비트를 표현할 수 있으며 (보통 $K = 32$), 따라서 합집합 계산은 N/K 번의 연산으로 처리된다.

또 다른 방법은 각 집합을 연결 리스트(linked list) 형태로 표현하는 것이다. 리스트는 변수 이름과 같은 정렬 가능한 키에 따라 정렬되어 있어야 한다. 이 방식에서 합집합은 두 리스트를 병합하면서 중복 항목을 제거하는 방식으로 계산되며, 이 연산은 병합 대상 집합들의 크기에 비례하는 시간이 소요된다.

요약하자면, 집합이 희소(sparse)한 경우 — 즉 평균적으로 N/K개보다 적은 원소를 포함하는 경우 — 정렬 리스트 방식이 점근적으로 더 빠르지만, 집합이 밀집(dense)되어 있다면 비트 배열 방식이 더 효율적이다.

시간 복잡도(Time Complexity). 반복(iterative) 기반 데이터플로 분석은 얼마나 빠를까? 크기 N의 프로그램은 흐름 그래프에서 최대 N개의 노드를 가지며, 최대 N개의 변수를 가진다고 하자. 따라서 각 *live-in*

집합(또는 *live-out* 집합)은 최대 N개의 원소를 가질 수 있다. 각 집합의 합집합 연산 하나를 계산하는 데는 $O(N)$ 시간이 걸린다.

for 루프는 각 노드마다 일정한 수의 집합 연산을 수행하므로, 노드가 $O(N)$개일 때 전체 수행 시간은 $O(N^2)$이다. **repeat** 루프의 각 반복에서 in/out 집합은 오직 증가만 할 수 있으며 절대 줄어들지 않는다. 이는 집합들이 서로 단조(monotonic) 관계에 있기 때문이다. 즉, 식 $\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$에서 $\text{out}[n]$이 커지면 $\text{in}[n]$도 반드시 커지고, 식 $\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$에서도 $\text{in}[s]$가 커지면 $\text{out}[n]$ 역시 커진다.

각 반복은 반드시 새로운 원소를 집합에 추가하지만, 모든 변수가 추가되어 더 이상 커질 수는 없다. in과 out 집합의 모든 원소의 총합은 $2N^2$ 이하이므로, 반복 루프가 그 이상 반복되지는 않는다. 따라서 최악의 경우 수행 시간은 $O(N^4)$이다. 그러나 노드들을 깊이 우선 탐색(DFS) 순서(알고리즘 17.5, p.390 참조)로 정렬하면 반복 루프의 반복 횟수는 보통 2~3번으로 줄어든다. 또한 대다수 경우 live 집합은 희소(sparse)하므로 실제 실행 시간은 보통 $O(N)$에서 $O(N^2)$ 수준이다.

최소 고정점(Least Fixed Points). 표 4.7은 식 4.3에 대한 두 가지 해(그리고 해가 아닌 예시 하나)를 보여준다. 이제 어떤 프로그램에 변수 d가 하나 더 존재하지만 이 코드 조각에서는 사용되지 않는다고 가정하자.

			X		Y		Z	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a	c	ac	cd	acd	c	ac
2	a	b	ac	bc	acd	bcd	ac	b
3	bc	c	bc	bc	bcd	bcd	b	b
4	b	a	bc	ac	bcd	acd	b	ac
5	a		ac	ac	acd	acd	ac	ac
6	c		c		c		c	

Figure 4.7: X and Y are solutions to the liveness equations; Z is not a solution.

해 Y에서는 변수 d가 루프를 따라 불필요하게 전달된다. 하지만 Y 역시 식 4.3을 만족한다는 점에서 X와 동일하게 유효한 해이다. 그렇다면 이는 무엇을 의미할까? d는 라이브한가, 라이브하지 않은가?

이에 대한 답은, 데이터플로 방정식의 모든 해는 *보수적 근사(conservative approximation)*라는 것이다. 만약 변수 a가 어떤 실행 경로에서 실제로 필요하다면, 식의 어떤 해에서도 a는 그 지점에서 live-out일 것이다. 그러나 그 반대는 성립하지 않는다. 즉, 우리가 d가 live-out이라고 계산했더라도 실제 실행에서 그 값이 사용된다는 보장은 없다.

이것이 괜찮은가? 이는 데이터플로 정보가 어디에 사용되느냐에 따라 달라진다. 라이브니스 분석에서는 어떤 변수가 라이브하다고 *추정된다면*, 그 값을 반드시 레지스터에 보관하려고 한다. 보수적 근사는 어떤 변수가 실제로는 데드(dead)임에도 불구하고 라이브하다고 잘못 판단할 수는 있지만, 그 반대 — 즉 실제 라이브한 변수를 데드라고 판단하는 일은 결코 없다. 보수적 근사의 결과로 컴파일된 코드가 실제보다 레지스터를 더 많이 쓸 수는 있지만, 정답은 항상 올바르게 계산된다.

반면, live-in 집합 Z처럼 식 4.3을 만족하지 않는 집합을 사용하면 문제가 발생한다. 예를 들어 Z가 b와 c가 동시에 라이브하지 않다고 잘못 판단한다면, 우리는 b와 c를 같은 레지스터에 할당할 것이다. 그 결과 필요한 레지스터 수는 최소가 되겠지만 *잘못된 프로그램*이 생성된다.

따라서 컴파일러 최적화에 사용되는 데이터플로 방정식은, 어떤 해이든 보수적인 정보를 제공하도록 구성되어야 한다. 부정확한 정보는 최적이 아닌 코드를 만들 수는 있어도, 잘못된 코드를 만들게 해서는 안 된다.

정리(Theorem). 식 4.3은 하나 이상의 해를 가진다.

증명. X와 Y는 모두 해이다.

정리(Theorem). 식 4.3의 모든 해는 해 X를 포함한다. 즉, 어떤 노드 n에 대해 해 X에서의 $\text{in}_X[n]$과 해 Y에서의 $\text{in}_Y[n]$이 주어졌을 때, $\text{in}_X[n] \subseteq \text{in}_Y[n]$이다.

증명. 연습문제 10.2 참조.

우리는 해 X를 식 4.3의 *최소 해(least solution)*라고 부른다. 더 큰 해는 더 많은 레지스터를 필요로 하므로(=비최적 코드), 우리는 항상 최소 해를 원한다. 다행히 알고리즘 4.4는 항상 최소 고정점을 계산한다.

정적 라이브니스 vs 동적 라이브니스 (Static vs. Dynamic Liveness). 변수는 "그 값이 미래에 사용될 가능성이 있을 때" 라이브하다고 말한다. 예를 들어 그래프 4.8에서 $b \times b$는 항상 음수가 아니므로, 조건식 $c \geq b$는 반드시 참이 된다. 따라서 노드 4는 절대 도달되지 않으며, 노드 2 이후에는 a의 값이 사용되지 않는다. 즉, a는 노드 2의 *live-out*이 아니다.

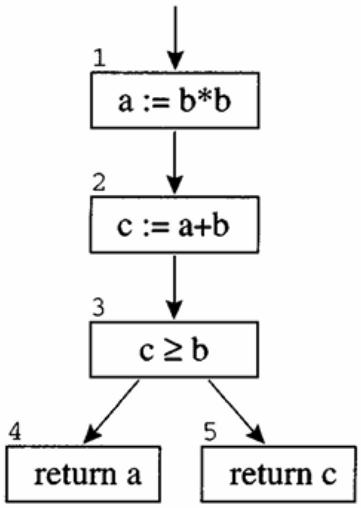


Figure 4.8: Standard static dataflow analysis will not take advantage of the fact that node 4 can never be reached.

그러나 방정식 4.3에 따르면 a는 노드 4의 *live-in*이며, 따라서 노드 3과 2의 *live-out*이기도 하다. 이 방정식들은 조건 분기가 실제로 어느 방향으로 진행되는지 알지 못하기 때문이다. 더 "똑똑한" 방정식이라면 a와 c를 동일한 레지스터에 할당하도록 허용했을 것이다.

$b \times b \geq 0$임을 증명하는 것은 가능하며, 컴파일러가 이런 산술적 항등식을 찾아내도록 만들 수도 있다. 그러나 어떤 컴파일러도 모든 프로그램의 제어 흐름을 100% 정확하게 이해할 수는 없다. 이는 정지 문제(halting problem)에서 유도되는 근본적인 수학적 정리 때문이다.

정리(Theorem). 어떤 프로그램 P와 입력 X에 대해, $P(X)$가 멈추는지(true) 아니면 무한 루프에 빠지는지(false)를 (자신은 멈추면서) 판별하는 프로그램 H는 존재하지 않는다.

증명. 그러한 프로그램 H가 존재한다고 가정하자. 그러면 우리는 다음과 같은 함수 F를 구성할 수 있다.

$$F(Y) = \text{if } H(Y, Y) \text{ then (while true do ()) else true}$$

이제 $F(F)$가 멈춘다면, $H(F, F)$는 true이므로 then 절이 실행되어 $F(F)$는 무한 루프에 빠지며 멈추지 않는다. 반대로 $F(F)$가 멈추지 않는다면, $H(F, F)$는 false이므로 else 절이 실행되어 $F(F)$는 멈춘다. 이는 모순이므로 그런 H는 존재하지 않는다.

따름정리(Corollary). 어떤 프로그램 X와 레이블 L이 주어졌을 때, 실행 중에 L이 도달되는지를 (항상 멈추면서) 판별하는 프로그램 $H'(X, L)$은 존재하지 않는다.

증명. 만약 그런 H'가 존재한다면, 이를 이용하여 H를 구성할 수 있다. 테스트하려는 프로그램에서 L을 프로그램의 끝에 위치시키고, 모든 `halt` 명령을 `goto L`로 대체하면 된다. 그러면 H'가 L 도달 여부를 판정함으로써 정지 여부를 알 수 있으므로 앞선 정리에 모순된다.

보수적 근사(Conservative Approximation). 이 정리는 우리가 특정 레이블이 실행 중에 도달되는지 *항상* 알아낼 수는 없다는 의미이지, *절대로* 알아낼 수 없다는 뜻은 아니다. 어떤 경우에는 더욱 정교한 기법을 사용하여 런타임 제어 흐름 정보를 계산할 수도 있다. 하지만 이런 특수한 기법도 모든 프로그램에서 항상 정확할 수는 없다.

이러한 근본적인 한계 때문에 컴파일러는 어떤 변수의 값이 실제로 필요할지 — 즉 그 변수가 진정으로 라이브인지 — 완벽히 알 수 없다. 따라서 우리는 *보수적 근사*를 사용해야 한다. 모든 조건 분기는 양쪽으로 모두 진행될 수 있다고 가정한다. 이에 따라 우리는 라이브니스를 동적으로 정의할 수도 있고, 그에 대한 정적 근사를 정의할 수도 있다.

동적 라이브니스(Dynamic Liveness). 변수 a는 어떤 실행 경로가 노드 n에서 a의 사용(use) 지점까지, 중간에 a의 정의(def)를 통과하지 않고 도달할 수 있다면, 노드 n에서 *동적으로 라이브*하다.

정적 라이브니스(Static Liveness). 변수 a는 제어 흐름 그래프 상에서 노드 n에서 a의 사용 지점까지, a의 정의를 통과하지 않는 경로가 *존재하기만 하면* 노드 n에서 *정적으로 라이브*하다.

분명히, 어떤 변수가 *동적으로 라이브*하다면 *정적으로도 라이브*하다. 최적화 컴파일러는 레지스터 할당 및 기타 최적화를 수행할 때 정적 라이브니스를 기준으로 삼는다. 왜냐하면 일반적으로 *동적 라이브니스*는 *계산할 수 없기* 때문이다.

인터피어런스 그래프(Interference Graphs). 라이브니스 정보는 컴파일러의 여러 종류의 최적화에 사용된다. 어떤 최적화에서는 흐름 그래프의 각 노드에서 정확히 어떤 변수들이 라이브한지를 알아야 한다. 라이브니스 분석의 가장 중요한 활용처 중 하나는 레지스터 할당이다. 즉, 임시 변수들의 집합 $a, b, c, \ldots$을 레지스터 $r_1, \ldots, r_k$에 배정해야 하는 상황이다. 두 변수 a와 b를 같은 레지스터에 배정할 수 없게 만드는 조건을 *인터피어런스(interference)*라고 부른다.

가장 흔한 종류의 인터피어런스는 라이브 구간이 겹치는 경우이다. 즉, 어떤 시점에서 변수 a와 b가 동시에 라이브하면, 두 변수는 같은 레지스터에 배치될 수 없다. 하지만 이 외에도 다른 형태의 인터피어런스가 존재한다. 예를 들어, 어떤 명령이 특정 레지스터 r_1에만 접근할 수 있다면, 변수 a가 그 명령에서 사용된다면 a와 r_1이 인터피어런스를 가진다고 볼 수 있다.

인터피어런스 정보는 행렬(matrix) 형태로 표현할 수 있으며, 그림 4.9a는 그래프 10.1에 등장하는 변수들의 인터피어런스를 x 표시로 나타낸 것이다. 또는 이 행렬을 무방향 그래프(undirected graph) 형태로 표현할 수도 있는데(그림 4.9b), 각 변수는 하나의 노드가 되고, 서로 인터피어하는 변수들 사이에는 간선이 연결된다.

Figure 4.9: Representations of interference.

MOVE 명령의 특별한 처리(Special treatment of MOVE instructions). 정적 라이브니스 분석에서는 `MOVE` 명령을 특별히 취급해야 한다. MOVE의 원본과 목적지 사이에 불필요한 인터피어런스가 생기지 않도록 주의해야 한다. 예를 들어, 다음 프로그램을 보자.

$$
\begin{aligned}
t &\leftarrow s \quad \text{(copy)} \\
&\vdots \\
x &\leftarrow \ldots s \ldots \quad \text{(use of } s\text{)} \\
&\vdots \\
y &\leftarrow \ldots t \ldots \quad \text{(use of } t\text{)}
\end{aligned}
$$

복사 명령 이후에는 s와 t가 모두 라이브이다. 일반적으로는 t가 정의되는 시점에 s가 여전히 라이브하므로 인터피어런스 간선 (s, t)를 추가하려고 할 것이다. 그러나 두 변수는 동일한 값을 담고 있으므로, 실제로는 s와 t를 다른 레지스터에 넣을 필요가 없다. 따라서 이 경우에는 (t, s) 간선을 추가하지 않는 것이 바람직하다. 물론 이후에 `MOVE`가 아닌 명령에서 t가 정의되고, 그 시점에 s가 여전히 라이브라면, 그때는 (t, s) 간선을 추가해야 한다.

따라서 각 새로운 정의 명령마다 인터피어런스 간선을 추가하는 규칙은 다음과 같다.

1. `MOVE`가 아닌 명령에서 어떤 변수 a가 정의되고, 그 시점의 *live-out* 변수들이 $b_1, \ldots, b_j$라면, 간선 $(a, b_1), \ldots, (a, b_j)$를 추가한다.
2. `a` $\leftarrow$ `c` 형태의 `MOVE` 명령에서 *live-out* 변수들이 $b_1, \ldots, b_j$라면, c와 같지 않은 모든 b_i에 대해 간선 (a, b_i)를 추가한다.

4.2 LIVENESS IN THE Tiger COMPILER

Tiger 컴파일러에서 흐름(flow) 분석은 두 단계로 이루어진다. 첫 번째 단계에서는 `Assem` 프로그램의 제어 흐름(control flow)을 분석하여 제어 흐름 그래프(control-flow graph)를 생성한다. 두 번째 단계에서는 이 제어 흐름 그래프 내에서 변수들의 라이브니스(liveness)를 분석하여 인터피어런스 그래프(interference graph)를 생성한다.

Graphs. 두 종류의 그래프를 모두 표현하기 위해, 우리는 `Graph`라는 추상 데이터 타입을 정의하자 (Program 4.10 참고).

```
signature GRAPH =
sig
  type graph
  type node

  val nodes: graph -> node list
  val succ: node -> node list
  val pred: node -> node list
  val adj: node -> node list
  val eq: node*node -> bool

  val newGraph: unit -> graph
  val newNode : graph -> node
  exception GraphEdge
  val mk_edge: {from: node, to: node} -> unit
  val rm_edge: {from: node, to: node} -> unit

  structure Table : TABLE
  sharing type Table.key = node

  val nodename: node->string  (*for debugging*)
end
```

Figure 4.10: The `Graph` abstract data type

함수 `newGraph`는 빈 유향 그래프(directed graph)를 생성한다; `newNode(g)`는 그래프 `g` 안에 새로운 노드를 추가한다. 어떤 노드 `m`에서 `n`으로 향하는 간선을 만들고 싶으면 `mk_edge(n, m)`을 사용한다; 그 후에는 `m`이 `succ(n)` 리스트에 포함되고 `n`이 `pred(m)`에 포함된다. 무방향 그래프를 다룰 때에는 `adj` 함수를 사용하는 것이 유용하다. 정의는 다음과 같다:

$$\mathrm{adj}(m) = \mathrm{succ}(m) \cup \mathrm{pred}(m)$$

간선을 삭제하려면 rm_edge를 사용한다. 노드 m과 n이 동일한 노드인지 테스트하려면 eq(m, n)을 사용한다.

어떤 알고리즘에서 그래프를 사용할 때에는, 각 노드가 어떤 의미 있는 것(예: 프로그램의 한 명령어)을 나타내도록 해야 한다. 노드에서 그것이 의미하는 실제 항목으로의 매핑을 만들기 위해 우리는 table을 사용한다. 다음과 같은 방식으로 정보 x를 노드 n과 연관시킬 수 있다:

```
Graph.Table.enter(mytable, n, x)
```

Control-flow graphs. Flow 구조체는 제어 흐름 그래프(control-flow graph)를 관리한다. 각 명령어(또는 기본 블록)는 제어 흐름 그래프의 하나의 노드로 표현된다. 만약 명령어 m 이후에 명령어 n이 실행될 수 있다면(점프에 의해서이거나 단순히 다음으로 이어지는 경우), 그래프에는 (m, n)이라는 간선(edge)이 존재한다.

```
structure Flow :
sig
  structure Graph
  datatype flowgraph =
    FGRAPH of {control: Graph.graph,
               def: Temp.temp list Graph.Table.table,
               use: Temp.temp list Graph.Table.table,
               ismove: bool Graph.Table.table}
end
```

플로우 그래프(flow graph)는 네 가지 구성 요소를 가진다:

- **control** 각 노드가 하나의 명령어(또는 기본 블록)를 나타내는 유향 그래프
- **def** 각 노드에서 정의되는 임시 변수들의 테이블 (명령어의 목적지 레지스터)
- **use** 각 노드에서 사용되는 임시 변수들의 테이블 (명령어의 소스 레지스터)
- **ismove** 각 명령어가 MOVE 명령인지 여부를 나타내는 플래그 (만약 def와 use가 동일하다면 삭제 가능)

The MakeGraph module. 이 모듈은 Assem 명령어 목록을 플로우 그래프로 변환한다.

```
structure MakeGraph:
sig
  val instrs2graph: Assem.instr list ->
                    Flow.flowgraph * Flow.Graph.node list
end
```

함수 instrs2graph는 명령어들의 리스트를 받아 플로우 그래프와, 각 명령어에 정확히 대응되는 노드들의 리스트를 반환한다. 플로우 그래프를 구성할 때, instrs의 jump 필드는 제어 흐름 간선을 구성하는 데 사용되며, src와 dst 필드에서 얻은 use와 def 정보는 flowgraph의 use 및 def 테이블을 통해 각 노드에 연결된다.

Information associated with the nodes. 플로우 그래프에서는 각 노드에 *use*와 *def* 정보를 연결해야 한다. 그리고 라이브니스 분석 알고리즘은 각 노드마다 *live-in*과 *live-out* 정보 역시 기억하고 싶어질 것이다. 우리는 이러한 모든 정보를 node 레코드 안에 직접 저장할 수도 있다. 이러한 방식은 잘 동작하며 효율성도 높다. 그러나 이 방식은 모듈성이 떨어질 수 있다. 결국 우리는 플로우 그래프에 대해 다른 종류의 분석도 수행하고 싶어질 것이며, 각 노드마다 다른 종류의 정보를 기억해야 할 수도 있다. 그때마다 데이터 구조(node)를 수정하게 되면, 이미 널리 사용되고 있는 인터페이스를 깨뜨릴 위험이 있다.

따라서 정보를 노드 내부에 저장하지 않고, 보다 모듈화된 접근법을 사용하는 것이 바람직하다. 즉, "그래프는 그래프일 뿐이며", 플로우 그래프는 그래프에 별도로 포장된 보조 정보(테이블, 또는 노드를 무엇이든지에 매핑하는 함수)를 더한 것이라고 보는 것이다. 마찬가지로, 그래프 위에서 동작하는 데이터플로 알고리즘은 노드 안에 정보를 수정할 필요가 없으며, 그 대신 자신이 가진 별도의 사설(private) 매핑 안에서 정보를 유지하면 된다.

효율성과 모듈성 사이에는 어느 정도의 절충(trade-off)이 존재할 수 있다. 단순한 포인터 순회(pointer traversal)를 통해 정보에 접근할 수 있다면, 해시 테이블이나 탐색 트리보다 빠를 수 있기 때문이다.

Liveness analysis. 컴파일러의 Liveness 모듈에는 interferenceGraph라는 함수가 있다. 이 함수는 플로우 그래프를 입력받아 두 가지 정보를 반환한다: 하나는 인터피어런스 그래프(igraph)이고, 다른 하나는 각 플로우 그래프 노드에 대해 그 노드에서 *live-out* 상태인 임시 변수들의 집합을 저장한 테이블이다.

```
structure Liveness:
sig
  datatype igraph =
    IGRAPH of {graph: IGraph.graph,
               tnode: Temp.temp -> IGraph.node,
               gtemp: IGraph.node -> Temp.temp,
               moves: (IGraph.node * IGraph.node) list}

  val interferenceGraph :
      Flow.flowgraph ->
      igraph * (Flow.Graph.node -> Temp.temp list)

  val show : outstream * igraph -> unit
end
```

The components of an igraph are:

- **graph** 인터피어런스 그래프 자체
- **tnode** `Assem` 프로그램의 임시 변수들을 그래프 노드로 매핑하는 함수
- **gtemp** 그래프 노드에서 다시 임시 변수로 되돌리는 역매핑 함수
- **moves** MOVE 명령들의 리스트. 이는 레지스터 할당기를 위한 힌트 역할을 한다. 만약 MOVE 명령이 (m, n)이면, 가능하다면 m과 n을 같은 레지스터에 배치하는 것이 좋다.

함수 `show`는 디버깅 용도로, 인터피어런스 그래프의 노드 목록과 각 노드에 인접한 노드들의 목록을 출력한다.

Data structure for liveness. `Liveness` 모듈의 구현에서는 각 플로우 그래프 노드에서 exit 시점에 라이브한 임시 변수들을 기억하는 자료구조가 유용하다.

```
type liveSet = unit Temp.Table.table * temp list
type liveMap = liveSet Flow.Graph.Table.table
```

플로우 그래프 노드 `n`이 주어졌을 때, 그 노드에서 live한 임시 변수들의 집합은 전역 `liveMap`에서 조회할 수 있다. 이 집합은 중복된 표현을 가진다: `table`은 빠른 membership 테스트를 위해 유용하고, `list`는 모든 라이브 변수를 나열할 때 유용하다.

완전한 `liveMap`을 계산하고 나면, 이제 인터피어런스 그래프를 구성할 수 있다. 즉, 플로우 그래프의 각 노드 `n`에서 새롭게 정의된 임시 변수 `d`가 있고, `liveMap`에 $\{t_1, t_2, \ldots\}$가 포함되어 있다면, 우리는 $(d, t_1), (d, t_2), \ldots$ 와 같은 인터피어런스 간선을 추가한다. MOVE 명령의 경우, 이러한 간선은 항상 정확하긴 하지만 최적은 아닐 수 있다 (221–222쪽에서 더 나은 방식이 설명된다).

Zero-length live ranges. 만약 새롭게 정의된 임시 변수가 정의 직후 라이브하지 않다면 어떻게 되는가? 다시 말해, 어떤 변수는 정의되긴 하지만 전혀 사용되지 않을 수도 있다. 이런 경우, 굳이 레지스터에 넣을 필요가 없으므로 다른 임시 변수들과 간섭하지 않을 것처럼 보인다. 그러나 정의 명령이 실제로 실행된다면 (어떤 부수효과를 위해서라도), 그 명령은 반드시 어떤 레지스터에 값을 기록할 것이다. 그리고 그 레지스터에는 다른 라이브 변수가 들어 있지 않아야만 한다. 따라서, 길이가 0인 라이브 구간(zero-length live range)도 자신과 겹치는 모든 라이브 구간과 간섭(interfere)한다.

Chapter 5

기호 실행

다음은 논문 Satisfiability Modulo Theories: A Beginner's Tutorial에서 발췌한 SMT의 배경이다[1].

위대한 사상가들은 오래전부터 연역적으로 추론할 수 있는 기계, 즉 일정한 가정들로부터 특정 결론이 논리적으로 따라오는지를 판정할 수 있는 기계를 만드는 꿈을 꾸어왔다. 이러한 기계가 가능한가라는 물음은 1928년 수학자 다비드 힐베르트(David Hilbert)에 의해 하나의 거대한 도전 과제로 공식적으로 제기되었으며, 그는 이를 "결정문제(Entscheidungsproblem)"라고 불렀다.

1936년에 처치(Church)와 튜링(Turing)은 이 문제가 일반적인 경우에는 해결 불가능(undecidable)하다는 것을 각각 증명하였다. 그럼에도 불구하고, 자동 추론(automated reasoning) 연구자들은 이 문제의 결정 가능한 특수한 경우를 해결하거나, 실제에서는 잘 작동하는 휴리스틱(heuristics)을 찾기 위한 노력을 계속해왔다.

그 과정에서 등장한 접근법이 바로 이론을 고려한 만족도 문제(Satisfiability Modulo Theories, SMT)이다. SMT는 결정 가능한 여러 이론(decidable theories)을 폭넓게 활용하여 표현력(expressive power)을 크게 높이면서도 결정 가능성(decidability)을 제공한다. 또한 SMT는 결정 불가능하거나 결정 가능성이 알려지지 않은 문제들에 대해서도 일부 질의를 수행할 수 있으며, 이때 강력한 휴리스틱 기법을 적용하여 실제로도 매우 효과적인 결과를 자주 보여준다.

- Church, A.: An unsolvable problem of elementary number theory. Am. J. Math. 58(2), 345–363 (1936)
- Turing, A.M.: On computable numbers, with an application to the entschei dungsproblem. J. Math. 58(345–363), 5 (1936)
- Hilbert, D., Ackermann, W.: Grundzüge der theoretischen Logik, Berlin 1928. Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete 27 (1938)

[1] Barrett, C. et al. (2025). Satisfiability Modulo Theories: A Beginner's Tutorial. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds) Formal Methods. FM 2024. Lecture Notes in Computer Science, vol 14934. Springer, Cham. https://doi.org/10.1007/978-3-031-71177-0_31

SMT를 활용한 응용 중 프로그램 검증 방법인 기호 실행[2] 을 살펴본다.

5.1 개요

신뢰할 수 있는 프로그램을 대규모로 생산하는 것은 오늘날의 도전적인 문제들에 컴퓨터를 적용하기 위한 근본적인 요구 조건 가운데 하나이다. 실제로 여러 기법이 사용되고 있고, 또 다른 기법들은 현재 연구의 초점이 되고 있다. 본 논문에서 보고하는 연구는 형식적인 명세가 주어지지 않은 경우에도 프로그램이 요구사항을 만족하도록 보장하는 데 초점을 둔다. 이 분야의 현재 기술은 기본적으로 테스트 기술이다. 즉, 프로그램이 처리할 것으로 기대되는 데이터에서 작은 표본을 선택하여 그것을 프로그램에 입력한다. 그 표본에 대해 프로그램이 올바른 결과를 내는 것으로 판단되면, 프로그램이 올바르다고 가정한다. 많은 최신 연구는 이러한 표본을 어떻게 선택할 것인가 하는 문제에 초점을 둔다.

프로그램을 형식적으로 분석하여 그 정확성을 증명하는 최근 연구는 매우 유망하며, 신뢰할 수 있는 프로그램을 만드는 궁극적인 기법으로 보인다. 그러나 이 영역에서의 실제 성과는 일상적으로 사용할 수 있는 도구 수준에는 미치지 못한다. 이론을 실제에 적용하는 데 필요한 근본적인 문제들이 가까운 시일 내에 해결될 것 같지는 않다.

프로그램 테스트와 프로그램 증명은 극단적인 두 대안으로 볼 수 있다. 테스트를 수행하는 동안에는, 프로그래머가 결과를 주의 깊게 검사함으로써 표본 입력에 대한 시험 실행이 올바르다는 점을 확신할 수 있다. 그러나 그 표본에 포함되지 않은 입력에 대한 실행이 올바른지는 여전히 의문으로 남는다. 반대로 프로그램 증명에서는, 프로그래머가 프로그램을 전혀 실행하지 않고도 모든 실행에 대해 프로그램이 명세를 만족함을 형식적으로 증명한다. 이를 위해 프로그래머는 올바른 프로그램 동작에 대한 정확한 명세를 제공하고, 그 다음 프로그램과 명세가 일관됨을 보이기 위해 형식적인 증명 절차를 따른다. 이 방법에 대한 신뢰는 명세를 작성하는 과정과 증명 단계를 구성하는 과정에서의 세심함과 정확성, 그리고 오버플로, 반올림 등의 기계 의존적 이슈에 얼마나 주의를 기울였는지에 달려 있다.

기호 실행(Symbolic Execution)은 이러한 두 극단 사이에 위치하는 실용적인 접근법이다. 단순한 관점에서 보면, 이 방법은 향상된 테스트 기법이다. 표본 입력 집합에 대해 프로그램을 실행하는 대신, 입력의 여러 부류에 대해 프로그램을 "기호적으로(symbolically)" 실행한다. 각 기호 실행 결과는 매우 많은 수의 일반적인 테스트 케이스와 동등한 효과를 낼 수 있다. 이러한 결과는 프로그래머가 기대하는 올바른 동작과 대조하여, 형식적으로 혹은 비형식적으로 검증할 수 있다.

각 기호 실행이 나타내는 입력 부류는 프로그램의 제어 흐름이 입력에 어떻게 의존하는지에 의해 결정된다. 만약 프로그램의 제어 흐름이 입력 변수와 전혀 독립적이라면, 하나의 기호 실행만으로도 프로그램의 모든 가능한 실행을 점검하기에 충분하다. 반대로 프로그램의 제어 흐름이 입력에 의존한다면, 경우 분석(case analysis)에 의존할 수밖에 없다. 모든 경우를 소진하기 위해 필요한 입력 부류의 집합이 실제로는 무한에 가깝게 되는 경우가 자주 있으므로, 이 방법 역시 기본적으로는 테스트 방법론이다. 그러나 이러한 입력 부류는 제어 흐름에 실제로 관여하는 입력들에 의해서만 결정되며, 대부분의 프로그램에 대해 기호 테스트는 일반적인 테스트보다 더 나은 결과를 더 수월하게 제공할 수 있을 것으로 기대된다.

[2] James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

5.2 기호 실행의 이상적 의미론

이 절에서는 프로그램의 기호 실행(symbolic execution)을 이상적인 관점에서 기술하고, 나중에 이러한 이상적 모델에 대한 근사물로 WHILE 프로그래밍 언어로 작성된 프로그램에 대한 기호 실행 시스템을 논의한다. 여기서 "이상적(ideal)"이라는 말을 사용하는 이유는 몇 가지가 있다.

1. 프로그램이 다루는 값이 정수뿐이라고 가정하고, 더 나아가 임의 크기의 정수만을 다룬다고 가정한다. 머신 레지스터 오버플로는 고려하지 않는다.

2. 많은(대부분의) 프로그램을 기호 실행했을 때 얻어지는 "실행 트리(execution tree)"(뒤에서 정의함)는 무한이다.

3. `IF` 문을 기호 실행하려면 정리 증명(theorem proving)이 필요하고, 심지어 그리 크지 않은 프로그래밍 언어에 대해서도 이러한 정리 증명은 기계적으로 불가능하다.

그럼에도 불구하고, 이러한 이상적 모델을 논의하는 것은 실제 컴퓨터 시스템이 수행하는 기호 실행을 평가할 수 있는 기준(standard)을 제공한다는 점에서 의미가 있다.

각 프로그래밍 언어는 실행 의미론(execution semantics)을 가진다. 실행 의미론은 프로그램 변수들이 나타낼 수 있는 데이터 객체, 언어로 작성된 문장들이 데이터 객체를 어떻게 조작하는지, 그리고 제어가 프로그램 문장들 사이를 어떻게 흐르는지를 기술한다. 이와는 별도로, 실제 데이터 객체를 직접 사용하지 않고 임의의 기호(symbol)로 표현하여 다루는 "기호 실행 의미론(symbolic execution semantics)"도 정의할 수 있다. 기호 실행은 정상 실행의 자연스러운 확장이며, 정상 실행은 기호 실행의 특수한 경우가 된다. 언어의 기본 연산자들에 대한 계산 정의를 확장하여, 기호 입력을 받아 기호식(symbolic formula)을 출력으로 내놓도록 만드는 식이다.

간단한 프로그래밍 언어 하나를 생각해 보자. 프로그램 변수는 모두 "부호 있는 정수(signed integer)" 타입이라고 하자. 단순 대입문, `IF` 문(`THEN`과 `ELSE` 절 포함), 레이블로 점프하는 `GO TO` 문, 그리고 입력을 얻는 수단(예: 프로시저 매개변수, 전역 변수, `read` 연산 등)을 포함시킨다. 산술식은 정수 덧셈 (+), 뺄셈 (−), 곱셈 (×) 같은 기본 정수 연산으로만 제한한다. `IF` 문에서 사용하는 부울 식(Boolean expression)은 "어떤 산술식이 음수가 아닌지"를 검사하는 단순한 형태, 즉 $\{\text{arith.expr.}\} \geq 0$ 꼴로 제한한다.

이제 이 단순 언어에 대해, 정상 실행 의미론은 이미 주어져 있다고 가정하고, 프로그램의 기호 실행을 설명한다. 기호 실행에서는 실행 의미론을 바꾸지만, 언어의 문법(syntax)이나 그 언어로 작성된 개별 프로그램은 바꾸지 않는다. 기호 데이터 객체(정수를 나타내는 기호)를 도입할 수 있는 유일한 지점은 프로그램의 입력 부분이다. 단순화를 위해, 프로그램이 새로운 입력 값을 필요로 할 때마다, 기호 입력을 기호들의 리스트 $\{\alpha_1, \alpha_2, \alpha_3, \dots\}$에서 하나씩 공급한다고 하자. 프로그램 입력은 결국 프로시저 매개변수, 전역 변수, `read` 문 등으로 프로그램 변수에 값으로 할당된다. 따라서 기호 입력을 처리하려면, 변수 값으로 정수 상수뿐 아니라 α_i들도 허용한다고 보면 된다.

대입문과 `IF` 문에 사용되는 산술식을 평가하는 규칙도 기호 값을 처리할 수 있도록 확장해야 한다. 정수, 미지 기호 집합 $\{\alpha_1, \alpha_2, \dots\}$, 괄호, 그리고 연산자 $+$, $-$, $\times$로 정상적인 방식으로 만들어지는 식은, 이 기호들에 대한 정수 다항식(integer polynomial)이다. 프로그램 변수가 α_i들에 대한 정수 다항식을 값으로 가질 수 있도록 허용하면, 대입문의 기호 실행은 자연스럽게 정의된다. 대입문의 오른쪽 식을 계산할 때,

필요하다면 변수 자리에 다항식 표현을 대입한다. 그 결과는 하나의 다항식(단순한 정수는 그 특수한 경우)이고, 이 다항식을 왼쪽 변수의 새 값으로 할당한다.

레이블로 가는 `GO TO` 문과 같은 제어 흐름은 정상 실행과 완전히 동일하게 동작한다. 즉, `GO TO` 문에서 해당 레이블이 붙은 문장으로 무조건 제어를 이전한다.

프로그램 실행의 "상태(state)"에는 보통 프로그램 변수들의 값과 현재 실행 중인 문장을 나타내는 문장 카운터(statement counter)가 포함된다. `IF` 문의 기호 실행을 정의하려면 여기에 "경로 조건(path condition, pc)"도 실행 상태에 포함해야 한다. pc는 입력 기호들 $\{\alpha_i\}$에 대한 부울 식이다. 프로그램 변수를 포함하지 않으며, 우리가 정의한 단순 언어에서는 $R \geq 0$ 또는 $\neg(R \geq 0)$ 꼴 식들의 논리곱(conjunction) 목록 형태이다. 여기서 R은 $\{\alpha_i\}$에 대한 다항식이다. 예를 들어 다음과 같다.

$$pc = (\alpha_1 \geq 0) \wedge (\alpha_1 + 2 \times \alpha_2 \geq 0) \wedge \neg(\alpha_3 > 0).$$

pc는 "어떤 입력들이 이 특정 실행 경로(path)를 따르기 위해 만족해야 하는 조건들"을 누적한 것이다. pc가 `true`로 초기화된 상태에서 기호 실행을 시작한다. 프로그램의 `IF` 문에서 여러 대안 경로 중 하나를 선택하기 위해 입력에 대한 가정을 세울 때마다, 그 가정들을 pc에 추가(논리곱)한다.

`IF` 문의 기호 실행은 정상 실행과 비슷하게 시작된다. 즉, `IF` 문에 붙어 있는 부울 식을, 변수 자리에 그 변수의 값을 대입하여 평가한다. 변수 값은 $\{\alpha_i\}$에 대한 다항식이므로, 그 조건식은 $R \geq 0$ 꼴이 되고, 여기서 R은 다항식이다. 이런 식을 q라고 부르자. 이제 현재 경로 조건 pc를 사용하여 다음 두 식을 만든다.

$$\text{(a)} \quad pc \Rightarrow q,$$
$$\text{(b)} \quad pc \Rightarrow \neg q.$$

명백히 pc 자체가 항상 거짓인 경우를 제외하면, 이 두 식 중 참이 될 수 있는 것은 많아야 하나이다. 만약 둘 중 정확히 하나만 참이라면, 정상 실행과 마찬가지로 `IF` 문 실행을 계속한다. (a)가 참이면 `THEN` 부분으로 제어를 넘기고, (b)가 참이면 `ELSE` 부분으로 제어를 넘긴다. pc를 만족하는 모든 일반 실행(normal execution)은, 이 기호 실행과 같은 분기를 택하게 된다. 즉, 모두 `THEN` 분기를 택하거나($pc \Rightarrow q$), 모두 `ELSE` 분기를 택한다($pc \Rightarrow \neg q$). 이런 경우 `IF` 문의 실행을 "비분기(nonforking) 실행"이라고 부른다.

더 흥미로운 경우는 (a)도 (b)도 참이 아닌 경우이다. 이때는 pc를 만족하면서 `THEN` 분기를 따르는 입력 집합이 적어도 하나 존재하고, pc를 만족하면서 `ELSE` 분기를 따르는 또 다른 입력 집합도 적어도 하나 존재한다. 두 분기가 모두 가능하므로, 모든 경우를 다루기 위한 유일한 완전한 방법은 두 제어 경로를 모두 탐색하는 것이다. 따라서 기호 실행은 두 개의 "병렬(parallel)" 실행으로 분기(fork)하도록 정의한다. 하나는 `THEN` 분기를 따라가고, 다른 하나는 `ELSE` 분기를 따라간다. 두 실행은 `IF` 문을 실행하기 직전의 동일한 계산 상태를 공유한 뒤, 그 이후로는 서로 독립적으로 진행된다. 이런 경우 `IF` 문 실행을 "분기(forking) 실행"이라고 부른다. 중요한 점은, 분기/비분기 속성은 `IF` 문 자체에 귀속되는 것이 아니라, 그 `IF` 문의 특정 실행에 귀속된다는 점이다. 같은 `IF` 문이라도 한 번의 실행에서는 분기 실행이 될 수 있고, 나중의 또 다른 실행에서는 비분기 실행이 될 수 있다.

`THEN` 분기를 택하면, 입력이 q (계산된 `IF` 조건식)를 만족한다고 가정한다. 이 정보는 pc에 다음과 같은

대입으로 기록한다.

$$pc := pc \wedge q.$$

마찬가지로 ELSE 분기를 택하면

$$pc := pc \wedge \neg q$$

로 갱신한다. pc가 "경로 조건(path condition)"이라고 불리는 이유는, 프로그램 안에서 특정한 하나의 제어 흐름 경로를 결정하는 조건들을 누적한 것이기 때문이다. IF 문의 각 분기 실행(forking execution)은, 어떤 분기를 택했는지에 따라 입력 기호들에 대한 조건 하나를 pc에 더해 준다. 비분기 실행인 경우에는 새로운 가정을 세우거나 필요로 하지 않기 때문에 pc는 변하지 않는다. pc는 처음 값이 true이고, pc에 대해 수행되는 유일한 연산이

$$pc := pc \wedge r \quad (\text{여기서 } r\text{은 } q \text{ 또는 } \neg q)$$

형태의 대입뿐이며, 그마저도 $(pc \wedge r)$이 만족 가능한 경우에만 수행되므로 pc가 거짓이 되어 버릴 수는 없다. $(pc \wedge r)$이 만족 가능하다는 것은, 논리식 $pc \Rightarrow \neg r$이 정리(theorem)가 아니라는 것과 동치이다.

5.2.1 기호 실행 예제

그림 5.1에 제시된 단순한 프로그램을 생각해 보자. 이 프로그램은 PL/I 스타일의 문법으로 작성되었고, 세 개의 값을 더하는 기능을 수행한다. 정수 입력 $1, 3, 5$에 대해, 그림 5.2과 같이 이 프로그램을 일반적으로 실행하면 결과 9를 계산한다. 그림 5.3에 자세히 나와 있는 기호 실행 결과를 통해, 임의의 세 정수 $\alpha_1, \alpha_2, \alpha_3$에 대해 이 프로그램이 이들의 합 $\alpha_1 + \alpha_2 + \alpha_3$을 계산한다는 사실이 이미 확립되었다.

이제 정수 X를 거듭제곱 Y만큼 올리는 프로그램을 나타낸, 보다 복잡한 예제를 그림 5.4에서 살펴보자. X와 Y에 대해 각각 기호 입력 α_1, α_2를 제공하면, 그 기호 실행은 그림 5.5과 같이 진행된다.

```
int sum(int a, int b, int c) {
  int x = a + b;
  int y = b + c;
  int z = x + y - b;
  return z;
}
```

Figure 5.1: sum 함수

After stmt	a	b	c	x	y	z
입력 직후	1	3	5	?	?	?
1	1	3	5	4	?	?
2	1	3	5	4	8	?
3	1	3	5	4	8	9
4 (**return**)	1	3	5	4	8	9

Figure 5.2: sum(1,3,5) 실행 시 변수값 변경

After stmt	a	b	c	x	y	z
입력 직후	α_1	α_2	α_3	?	?	?
1: $x = a + b$	α_1	α_2	α_3	$\alpha_1 + \alpha_2$	?	?
2: $y = b + c$	α_1	α_2	α_3	$\alpha_1 + \alpha_2$	$\alpha_2 + \alpha_3$	?
3: $x = x + y - b$	α_1	α_2	α_3	$\alpha_1 + \alpha_2$	$\alpha_2 + \alpha_3$	$\alpha_1 + \alpha_2 + \alpha_3$
4: `return z`	α_1	α_2	α_3	$\alpha_1 + \alpha_2$	$\alpha_2 + \alpha_3$	$\alpha_1 + \alpha_2 + \alpha_3$

Figure 5.3: `sum`$(\alpha_1, \alpha_2, \alpha_3)$ 기호 실행: $pc =$ `true`로 시작하여 `sum`은 항상 $\alpha_1 + \alpha_2 + \alpha_3$를 반환한다.

```
1  int popwer(x, y) {
2    z = 1;
3    j = 1;
4
5    while ( y >= j ) {
6      z = z * x;
7      j = j + 1;
8    }
9
10   return z;
```

Figure 5.4: `power` 함수

5.2.2 기호 실행 트리

기호 실행 동안 따라가게 되는 실행 경로들을 "실행 트리(execution tree)"로 생성할 수 있다. 각 실행된 문장마다 하나의 노드를 대응시키고(노드에는 문장 번호를 레이블로 붙인다), 문장 사이의 실행 단계를 해당 노드들을 연결하는 방향 있는 선으로 연결해 대응한다. 분기하는 `IF` 문의 실행마다, 그에 대응하는 노드는 두 개의 나가는 에지를 가지며, 각각 `THEN` 분기(참)와 `ELSE` 분기(거짓)를 나타내도록 "T"와 "F" 레이블을 붙인다. 또한 각 노드에는 현재 실행 상태 전체—즉, 변수 값들, 문장 카운터, 그리고 경로 조건인 pc—를 함께 대응시킨다. `power`(α_1, α_2) (그림 5.4, 5.5)에 대한 실행 트리는 그림 5.6에 제시되어 있다.

이와 같이 기호 실행으로부터 얻어지는 트리는 다음과 같은 성질을 가진다.

1. 각 실행 트리의 단말 리프(하나의 실행이 끝나는 지점)는 항상 어떤 "진짜" 입력값들과 연결된다. 즉, 그 리프의 pc를 만족하는 구체적인 입력을 프로그램에 넣고 정상적으로 실행하면, 기호 실행에서 따라간 것과 똑같은 경로 (같은 문장 실행 순서)를 따라가게 되는 입력이 적어도 하나 존재한다. 이는 pc가 결코 항상 거짓이 되지 않는다는 뜻이다. 이 사실에 대한 논증은 앞에서 간단히 설명하였다.

2. 서로 다른 두 단말 리프에 대응하는 경로 조건 pc들은 항상 서로 겹치지 않는다. 다시 말해, 두 리프의 경로 조건 pc_1, pc_2에 대해 $pc_1 \wedge pc_2$는 항상 거짓(만족 불가능, Unsatisfiable)이다, 즉 $\neg(pc_1 \wedge pc_2)$이다.

   이를 실행 트리 관점에서 보면, 공통 루트에서 출발해 두 단말 리프까지 가는 두 경로를 따라 올라가면, 이 두 경로가 처음으로 갈라지는 분기 노드가 정확히 하나 있다. 그 분기에서 어떤 조건 q는 한쪽 경로의 pc에 추가되고, 반대쪽 경로의 pc에는 $\neg q$가 추가된다. 따라서, 두 경우 모두 pc는 끝까지 유지되지만, 서로 다른 두 단말 리프에 대응하는 서로 다른 경로 조건식을 갖게 된다. 그 결과, 두 단말 리프는 서로 양립할 수 없는(함께 참이 될 수 없는) 조건을 가지게 되고, 서로 다른 두 단말 리프는 서로 겹치지 않는 입력 집합을 나타낸다.

초기 상태 및 while 반복문 직전까지

After stmt	j	x	y	z	pc
입력 직후	?	α_1	α_2	?	*true*
2: $z = 1$	?	α_1	α_2	1	*true*
3: $j = 1$	1	α_1	α_2	1	*true*

5번 문장 `while ( y >= j )`에서 1차 분기

1. 조건식 평가: $y \geq j \;\Rightarrow\; \alpha_2 \geq 1$.
2. 현재 경로 조건 $pc = true$에서 다음을 검사한다.
   (a) $true \vdash \alpha_2 \geq 1$?
   (b) $true \vdash \neg(\alpha_2 \geq 1)$?
3. 어느 쪽도 항상 참은 아니므로, 양쪽을 모두 기호 실행한다.

Case $\neg(\alpha_2 \geq 1)$ **(즉** $\alpha_2 < 1$**)** 이 경로에서는 $\alpha_2 < 1$일 때 $Z = 1$을 반환하고 종료한다.

After stmt	j	x	y	z	pc
5 (`while` 반복문 밖으로)	1	α_1	α_2	1	$true \wedge \neg(\alpha_2 \geq 1)$
10: `return z`	1	α_1	α_2	1	$true \wedge \neg(\alpha_2 \geq 1)$

Case $\alpha_2 \geq 1$

After stmt	j	x	y	z	pc
5 (`while` 반복문 안으로)	1	α_1	α_2	1	$\alpha_2 \geq 1$
6: $z = z * x$	1	α_1	α_2	α_1	$\alpha_2 \geq 1$
7: $j = j + 1$	2	α_1	α_2	α_1	$\alpha_2 \geq 1$

다시 5번 while 문장의 조건을 검사 $y \geq j \Rightarrow \alpha_2 \geq 2$가 되며, $pc = (\alpha_2 \geq 1)$로 다시 한 번 분기.

1. **Case** $\alpha_2 \geq 1 \wedge \neg(\alpha_2 \geq 2)$ **(즉** $\alpha_2 = 1$**)**에서는 `while` 반복문을 빠져나와 $z = \alpha_1$을 반환
2. **Case** $\alpha_2 \geq 2$: 같은 패턴으로 루프 본문을 다시 실행하며, j와 pc가 계속 증가/강화됨. α_2가 임의의 큰 정수일 수 있으므로, 전체 기호 실행 트리는 무한히 확장될 수 있음.

Figure 5.5: `power`(α_1, α_2) 기호 실행

이러한 성질의 의미는 그림 5.7의 예제를 통해 더 분명해진다. 간단함을 위해 `for` 문을 사용했는데, 이는 앞에서 사용한 표기와 맞추기 위해 `while` 루프로 쉽게 전개할 수 있다. `twoloops` (그림 5.7)에 대한 실행 트리는 그림 5.8에 나와 있다.

두 번째 루프의 4번 문장은, 문법적으로는 첫 번째 루프의 2번 문장과 똑같이 분기할 수 있는 구조를 가진다. 하지만 2번 문장에서 분기할 때 만들어 둔 경로 조건 pc 안에 이미 N에 대한 정보가 충분히 쌓여 있기 때문에, 4번 문장의 조건은 pc만 보고도 참/거짓을 결정할 수 있다. 그 결과 4번 문장에서는 새로운 fork가 생기지 않고, 항상 한쪽 분기만 따라가게 된다. `twoloops` (그림 5.7)에 대한 실행 트리는 그림 5.8을 통해 4번 문장의 for 루프에 대한 fork가 없고 선형 구조임을 확인할 수 있다.

5.2.3 Commutativity

앞에서 정의한 이 단순 정수 언어의 기호 실행은 흥미로운 교환 성질을 만족한다. 기호 $\{\alpha_i\}$를 구체적인 정수 $\{j_i\}$로 치환(instantiation)하는 연산과 프로그램을 실행하는 연산이 서로 순서를 바꾸어도 결과가 같다는 것이다.

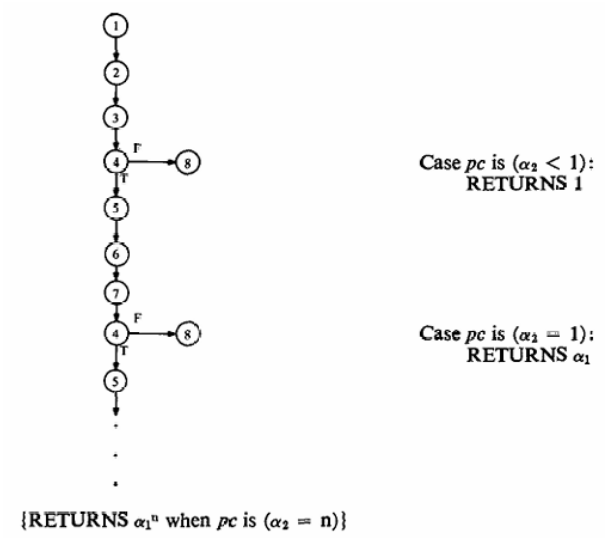


Figure 5.6: power($alpha_1$, $alpha_2$) 함수 호출을 기호 실행

```
1 void twoloops(n) {
2  for (j = 1; j < n; j++)
3    (body of statements)
4  for (k = 1; k < n; j++)
5    (body of statements)
6 }
```

Figure 5.7: `twoloops` 함수

좀 더 구체적으로 말하면, 입력으로 특정 정수 집합 $\{j_i\}$를 주어 프로그램을 일반적으로 실행하는 경우(먼저 α_i들을 j_i들로 치환한 뒤 실행하는 경우)와, 프로그램을 먼저 기호적으로 실행한 다음 그 기호 결과에서 α_i들을 j_i들로 치환하는 경우의 결과가 서로 같다.

여기서 "기호 결과를 치환(instantiating the symbolic results)한다"는 의미는 다음과 같다. 실행 트리의 각 단말 리프에 대해, 그 노드에 기록된 모든 프로그램 변수 값들과 경로 조건 pc에서 α_i들을 j_i로 치환한다. 그 후, 치환된 pc가 참이 되는 단말 노드의 변수 값들이 바로 우리가 얻고자 하는 "결과"가 된다.

이 교환 성질은 그림 5.9과 같이 도식적으로 표현할 수 있다. 여기서 P는 프로그램을 나타내고, $E(P(X))$는 입력 X에 대해 프로그램 P를 실행한 결과를 의미하며, K는 구체적인 정수 입력들의 집합이다.

물론 이러한 교환 관계가 성립하기 때문에 기호 실행이 의미 있는 기법이 되는 것이다. 기호 실행은 일반 실행과 정확히 같은 효과를 포착한다. 기호 실행은 단순히 임의로 정의된 다른 실행 의미가 아니라, 기존의 실행 의미를 자연스럽게 확장한 것이라고 볼 수 있다. 산술(arithmetic)과 대수(algebra)의 관계와 마찬가지로, 프로그램 연산자가 지시하는 구체적인 산술 계산을 적절한 대수식으로 일반화하고 "미룬(delayed)" 형태로 표현하는 것이 기호 실행의 역할이다.

5.2.4 프로그램 정확성, 증명, 기호 실행

Floyd가 제시한 방법을 사용하여 프로그램의 정당성(correctness)을 증명하려면, 프로그래머는 프로그램과 함께 "입력 조건(input predicate)"과 "출력 조건(output predicate)"을 제공해야 한다. 이 두 조건은 프로

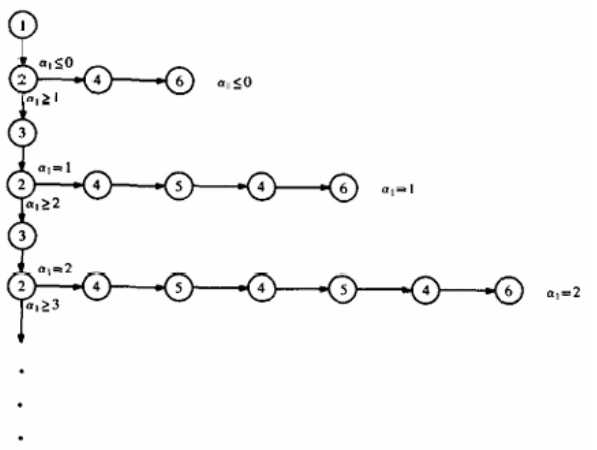


Figure 5.8: `twoloops` 함수 실행 트리

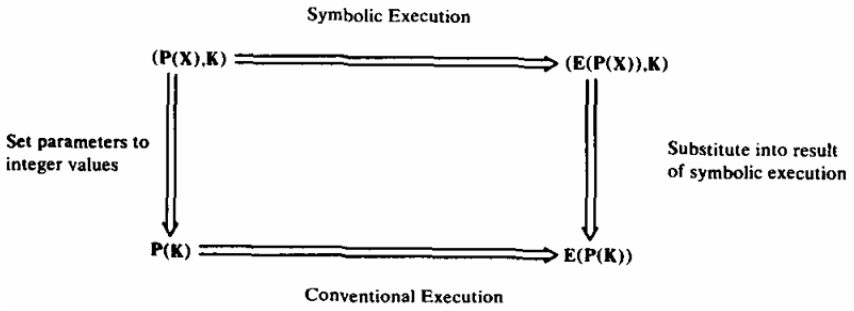


Figure 5.9: 기호 실행과 일반 실행 사이의 교환 관계를 나타내는 도식

그램의 "올바른" 동작을 정의하며, 모든 입력 값이 입력 조건을 만족할 때, 프로그램이 (정상적으로) 만들어 내는 결과가 출력 조건을 만족하면 그 프로그램을 올바르다고 본다. Floyd는 프로그램과 그 입출력 조건이 서로 일관(consistency)을 이루는지 점검하는 방법, 즉 정당성을 증명하는 방법을 제시하였다.

Floyd의 증명 방법에서 한 단계인 "검증 조건(verification condition)의 생성"은 프로그램 경로를 기호적으로 실행함으로써 매우 간단하게 수행할 수 있다. Deutsch는 자신의 대화형 프로그램 검증기(interactive program verifier)에서 이 증명 기법을 활용하면서, 독자적으로 기호 실행(symbolic execution) 개념을 발전시켰다.

이 기법을 설명하는 가장 단순한 방법은, 프로그램에 논리식을 연결하기 위해 사용하는 세 개의 보조 언어 구문, 즉 `ASSUME`, `PROVE`, `ASSERT`를 사용하는 것이다. 이 세 구문은 모두 `ASSERT(X>0)`과 같이 괄호 안에 부울 식(Boolean formula)을 인자로 가진다. 이 식에 등장하는 자유 변수들은 모두 프로그램 변수라고 가정한다. ASSUME 문장의 동작은 다음과 같다. 이 문장이 실행되면, 현재 프로그램 변수 값으로 식 B를 평가하고, 그 평가 결과를 경로 조건(path condition)에 논리곱으로 추가한다. 즉 다음과 같이 수행된다.

$$pc := pc \wedge \text{value}(B).$$

`PROVE(B)` 문장은 다음 식을 만들고

$$pc \Rightarrow \text{value}(B)$$

이 식이 정리(theorem)인지 증명하려 시도한 뒤, 그 결과에 따라 참(true) 또는 거짓(false)을 출력한다. 아래에서 PROVE를 사용하는 방식에서는, 이 식들이 바로 Floyd의 검증 조건(verification conditions)이 될 것이다. ASSERT 구문은 이후에 상황에 따라 ASSUME 또는 PROVE로 해석하여 사용할 것이다.

프로그램의 정당성을 정의하는 초깃값/최종값 조건(initial/final predicates) 외에도, Floyd의 방법에서는 프로그램의 여러 지점에 추가적인 "귀납적 조건(inductive predicates)"을 부착해야 한다. 보통은 프로그램의 각 루프 내부에 적어도 하나의 조건을 두도록 설계하지만, 논리적으로는 각 조건 사이의 프로그램 구간(해당 조건에서 다음 조건까지의 경로)이 유한 길이가 되도록 아무 방식으로나 배치해도 된다. 이 귀납적 조건들은 일반적인 귀납(유도) 논증을 가능하게 해 주며 (자세한 설명은 Floyd 연구 참고), 프로그램 전체의 정당성 증명을 유한 개의, 길이가 유한한 경로들에 대한 정당성 증명으로 환원시켜 준다.

이제 입력, 출력, 귀납 조건을 모두 프로그램 안의 적절한 위치에 ASSERT 문장으로 삽입하여 연결했다고 가정하자 (예를 들어, 입력 조건은 프로그램의 첫 문장으로 ASSERT를 두어 표현하는 식이다). 그러면 우리는 프로그램에 대해 고정된 집합의 경로를 가지게 된다. 각 경로는 ASSERT 문장에서 시작하여 ASSERT 문장에서 끝나며, 각 경로는 모두 정당성을 증명해야 한다. 즉, 경로의 시작 지점에서 *초기 조건*을 만족하는 변수값 집합을 임의로 택했을 때, 그 경로를 따라 프로그램을 실행한 결과로 얻어진 변수값들이 항상 *종료 조건*을 만족함을 보여야 한다.

각 경로의 정당성은 다음과 같이 그 경로를 기호 실행함으로써 증명할 수 있다.

1. 경로 시작 부분의 ASSERT를 ASSUME로 바꾸고, 경로의 끝 부분에 있는 ASSERT를 PROVE로 바꾼다.
2. 경로 조건을 $pc = \mathsf{true}$로 초기화하고, 모든 변수에 서로 다른 기호 값(예: $\alpha_1, \alpha_2, \ldots$)을 할당한다.
3. 경로를 기호 실행한다. 이때 경로 상에서 아직 분기가 결정되지 않은 IF 문 실행을 만나면, 해당 경로 정의에 맞추어 `go true` 또는 `go false`를 선택하여 미리 정해진 그 경로를 계속 따르도록 한다.
4. 경로 끝의 PROVE 문장이 참(true)이면 그 경로는 정당하고, 거짓(false)이면 정당하지 않다.

기호 실행이 5.2.3 절에서 논의한 교환(가환) 성질을 만족한다고 가정하면, 이 방법이 올바른 증명 기법이라는 점은 비교적 직관적으로 이해할 수 있다. 프로그램에 대한 증명은, 경로 시작점에서의 변수값들(우리가 기호 이름을 붙인 값들)을 기준으로 이루어진다. 경로 조건 pc는 그러한 초기값들에 대해 우리가 세운 모든 가정을 누적한다. 경로 시작 위치에서 실행되는 첫 ASSUME 문장은, 초기값들이 만족해야 할 조건을 pc에 기록한다. (즉, "경로 시작 조건을 만족하는 모든 변수값 집합에 대해, 해당 경로를 따라 실행한 결과가 경로 끝의 조건을 만족함을 보여야 한다"는 요구를 반영한다.) 대입문 실행의 결과로 계산되는 수식들은, 경로 시작점의 값들을 인자로 하는 함수 형태로 프로그램 변수들의 갱신된 값을 기록한다. 아직 분기가 결정되지 않은 IF 문 실행에서 하는 작업은, 해당 경로를 실제로 따라가기 위해 추가로 필요한 가정들을 역시 경로 시작값들에 대한 식으로 pc에 기록하는 것이다.

마지막으로, 경로 끝에서 실행되는 PROVE 문장은 다음과 같은 정리 후보(검증 조건)를 구성한다.

$$pc \wedge (\text{시작 조건}) \Rightarrow (\text{끝 조건이 현재 변수값에 대해 성립}).$$

이는 곧 "시작 조건이 만족되었고, 우리가 이 특정 경로를 따라 실행했다(pc에 기록됨)고 가정할 때, 경로 끝에서의 현재 변수값들이 종료 조건을 만족하는가?"라는 질문을 논리식의 형태로 표현한 것이다.

기호 실행 시스템은, PROVE 문장과 경로를 나열하고, IF 문에서 경로 선택을 강제로 지정해 주는 관리용 컨트롤러를 추가함으로써, 정당성 증명을 시도하게 만들 수 있다. 실제로 이를 구현한 시스템이 있고, 정당성 증명 기법에 대한 연구를 수행하는 도구로 사용되었다.

사실 정당성 증명(program correctness proof)과 기호 실행(symbolic execution)의 개념은, 개념 자체에서도, 이를 수행하는 데 필요한 도구 측면에서도 서로 밀접하게 연관되어 있다. 예를 들어, 어떤 프로그램에 Floyd 스타일의 입출력 조건을 부여했다고 하자. 입력 조건은 프로그램 시작 부분에 ASSUME 문장으로, 출력 조건은 프로그램 끝 부분에 PROVE 문장으로 넣는 것이다. 여기서 정의한 ASSUME의 의미에 따르면, 이 초기 ASSUME은 기호 실행이든 정당성 증명이든 이후의 모든 분석을 "초기 조건을 만족하는 값들"로만 제한하게 만든다. 또 정당성 증명을 위해 필요한 방식으로 PROVE를 정의해 두면, 이 정의는 기호 실행을 이용한 프로그램 테스트에도 매우 유용하게 사용된다. 테스트(기호 테스트이든, 일반 테스트이든)에서는 프로그램이 내놓은 출력을 검사하고 그 정당성을 판단해야 한다. 만약 우리가 정당성 기준을 출력 조건(output predicate)으로 형식화하여 프로그램 끝에 PROVE 문장으로 넣을 수 있다면, 기호 실행기는 테스트 결과가 올바른지에 대한 검사를 자동으로 수행할 수 있다.

PROVE 구문의 정의와 관련된 한 가지 추가 설명으로, 몇몇 언어(예: Algol W)는 프로그램 안에 ASSERT 구문을 두고 실행 시간에 그 조건을 검사하는 기능을 이미 제공한다. 이러한 구문은 보통 다음과 같이 쉽게 구현할 수 있다.

ASSERT(B) 를 IF $\neg B$ THEN SIGNAL ERROR 의 축약 표현으로 간주한다.

이는 일반적인 IF 문이므로, 그 기호 실행을 고려할 수 있다. 프로그램이 올바르다면, 이 IF 문은 항상 거짓 분기(ELSE 쪽)로만 실행 가능해야 한다. IF 문의 기호 실행 정의에 따르면, 이는 오직 다음이 항상 성립할 때만 가능하다.

$$pc \Rightarrow \text{value}(B).$$

이것이 바로 PROVE(B)가 증명하려고 하는 것과 정확히 같은 식이다.

어떤 프로그램에 대해 기호 실행 트리가 유한하고, 또 그 프로그램의 정당성 기준이 입출력 조건으로 명시되어 있다고 하자. 이 경우, 앞에서 설명한 방식으로 수행하는 *완전한 기호 실행*과 *정당성 증명 과정*은 사실상 동일한 과정이다. 이때는 귀납적 조건이 필요 없으며, 증명이 필요한 경로 집합은 단지 입력 조건에서 시작하여 출력 조건에 도달하는 경로들이고, 이는 곧 유한한 실행 트리가 기술하는 경로 집합과 정확히 일치한다.

반대로, 실행 트리가 무한한 프로그램의 경우, 기호 테스트(symbolic testing)는 완전할 수 없고, 절대적인 의미의 정당성 증명도 얻을 수 없다. 실행 트리의 무한한 부분에 대해 어떤 형태로든 귀납(induction)을 도입하여 기호 실행을 "무한 가지(branches) 위로까지" 확장해야만 모든 경우에 대한 정당성 증명이 가능해진다. 이는 곧 Floyd의 정당성 증명 기법이 하는 일과 정확히 같다. 루프 내에 두는 귀납적 조건들은, 실행 트리의 무한한 가지들 위로 기호 실행이 "논리적으로 실행되게" 하는 데 필요한 귀납적 도움을 제공한다. 또 다른, 유사하지만 다른 접근법도 있다. Topor와 Burstall이 제안한 방법은 실행 트리의 무한 가지들에 대해 "실행"을 가능하게 하고, 정당성 증명을 제공하기 위한 귀납적 보조 장치를 도입한다.

마지막으로, 정당성 증명을 할 때 필요한 "술어(predicate)를 다루는 능력"과 단순히 기호 실행으로 테

스트를 할 때 필요한 능력 사이에는 중요한 차이가 있다.

우선, 사용자가 별도의 술어를 추가하지 않은 순수한 기호 실행만 생각해 보면, 경로 조건 pc와 그때그때 증명해야 하는 식들은 전부 프로그램 언어와 프로그램 코드 자체에 의해 자동으로 정해지는 것들이다. 즉, 어떤 식이 나오는지는 언어의 연산과 문장들에 의해 문법적 · 의미론적으로 결정되는 셈이다.

하지만 정당성 증명에서 등장하는 술어들의 의미는 다르다. 이들은 언어 차원에서 자동으로 생기는 식이 아니라, 그 프로그램이 풀고자 하는 "문제 영역"에서 온다. 다시 말해, 이 술어들은 코드 문법이 아니라 도메인 지식과 요구사항을 표현하는 역할을 한다.

이 차이 때문에, 적어도 단기적으로는 프로그램 전체에 대한 일반적인 검증(program verification) 기법보다 기호 실행을 이용한 테스트 기법이 더 실질적으로 활용 가능한 기술이라고 우리는 판단한다.

5.3 WHILE 프로그램 기호 실행

본 문서는 WHILE 프로그래밍 언어에 대한 기호 실행(Symbolic Execution)의 형식적 명세를 제시한다. 기호 실행은 프로그램 분석 기법 중 하나로, 구체적인 입력값 대신 기호적(symbolic) 값을 사용하여 프로그램의 실행 경로를 탐색하고 경로 조건(path condition)을 수집하여 SMT 솔버를 통해 프로그램의 특성을 검증한다. 이를 통해 다음을 달성할 수 있다:

- 프로그램의 모든 가능한 실행 경로 탐색
- 특정 경로에 도달하기 위한 입력 조건 생성
- 버그나 취약점 발견
- 테스트 케이스 자동 생성

Definition 5.3.1 (기호 상태)**.** *기호 상태(Symbolic State)는 $\sigma : Vars \to Z3Var$과 ϕ : 경로 조건 (Path Condition)로 구성된 $\langle \sigma, \phi \rangle$이다. 여기서 σ는 프로그램 변수를 Z3 변수(AST)로 매핑하는 환경이고, ϕ는 현재 실행 경로의 조건을 나타내는 논리식이다.*

Definition 5.3.2 (Z3 변수)**.** *각 프로그램 변수 x는 실행 중 여러 버전을 가질 수 있으며, SSA(Static Single Assignment) 형태로 표현된다:*

$$x_0, x_1, x_2, \ldots$$

각 버전은 32비트 비트벡터(bit-vector)로 표현된다.

기호 실행 관계를 $\langle \sigma, \phi, c \rangle \Rightarrow \langle \sigma', \phi' \rangle$로 표기하며, 이는 명령 c를 환경 σ와 경로 조건 ϕ 하에서 실행하여 새로운 환경 σ'와 경로 조건 ϕ'를 생성함을 의미한다.

Rule 5.3.3 (Skip)**.**

$$\frac{}{\langle \sigma, \phi, \texttt{skip} \rangle \Rightarrow \langle \sigma, \phi \rangle} \tag{5.1}$$

`skip` 명령은 상태를 변경하지 않는다.

Rule 5.3.4 (대입문 (Assignment))**.**

$$\frac{[\![e]\!]_\sigma = e' \quad x_{new} = \mathit{fresh}() \quad \phi' = \phi \wedge (x_{new} = e')}{\langle \sigma, \phi, x := e \rangle \Rightarrow \langle \sigma[x \mapsto x_{new}], \phi' \rangle} \tag{5.2}$$

변수 x에 식 e를 대입할 때:

- *식 e를 현재 환경에서 기호적으로 평가하여 e'를 얻는다*
- *새로운 SSA 버전 x_{new}를 생성한다*
- *경로 조건에 $x_{new} = e'$ 제약을 추가한다*
- *환경을 업데이트한다*

Rule 5.3.5 (순차 실행 (Sequence))**.**

$$\frac{\langle \sigma, \phi, c_1 \rangle \Rightarrow \langle \sigma', \phi' \rangle \quad \langle \sigma', \phi', c_2 \rangle \Rightarrow \langle \sigma'', \phi'' \rangle}{\langle \sigma, \phi, c_1; c_2 \rangle \Rightarrow \langle \sigma'', \phi'' \rangle} \tag{5.3}$$

순차 실행은 첫 번째 명령의 결과 상태에서 두 번째 명령을 실행한다.

Rule 5.3.6 (조건문 (If-Then-Else))**.**

$$\frac{[\![e]\!]_\sigma = b \quad \langle \sigma, \phi \wedge b, c_1 \rangle \Rightarrow \langle \sigma_1, \phi_1 \rangle \quad \langle \sigma, \phi \wedge \neg b, c_2 \rangle \Rightarrow \langle \sigma_2, \phi_2 \rangle \quad \sigma' = \mathit{merge}(b, \sigma, \sigma_1, \sigma_2)}{\langle \sigma, \phi, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rangle \Rightarrow \langle \sigma', \phi_1 \vee \phi_2 \rangle} \tag{5.4}$$

조건문 처리 시:

1. *조건식 e를 기호적으로 평가하여 b를 얻는다*
2. *Then 분기: 경로 조건 $\phi \wedge b$ 하에서 c_1을 실행*
3. *Else 분기: 경로 조건 $\phi \wedge \neg b$ 하에서 c_2를 실행*
4. *두 분기의 결과를 병합(merge)한다*

병합 함수(Phi 노드): *각 변수 x에 대해, 새로운 버전 x_{new}를 생성하고:*

$$x_{new} = \mathit{ite}(b, x_{\sigma_1}, x_{\sigma_2}) \tag{5.5}$$

여기서 $\mathit{ite}(b, v_1, v_2)$는 if-then-else 연산자로, b가 참이면 v_1, 거짓이면 v_2를 선택한다.

반복문

Rule 5.3.7 (반복문 언롤링 (While Loop Unrolling))**.**

$$\texttt{while}\ e\ \texttt{do}\ c \equiv \underbrace{\texttt{if}\ e\ \texttt{then}\ (c; \texttt{if}\ e\ \texttt{then}\ (c; \ldots)\ \texttt{else skip})}_{n\text{번 언롤링}} \tag{5.6}$$

반복문은 유한한 횟수 n만큼 언롤링하여 처리한다:

$$unroll(n, \texttt{while}\ e\ \texttt{do}\ c) = \begin{cases} \texttt{skip} & \text{if } n = 0 \\ \texttt{if}\ e\ \texttt{then}\ (c; unroll(n-1, \texttt{while}\ e\ \texttt{do}\ c))\ \texttt{else skip} & \text{if } n > 0 \end{cases}$$

이 방법의 한계:

- *n회 이상 반복하는 경로는 탐색하지 못함*
- *언롤링 횟수가 너무 크면 경로 폭발(path explosion) 발생*

Rule 5.3.8 (읽기 (Read))**.**

$$\frac{x_{new} = fresh()}{\langle \sigma, \phi, \texttt{read}(x) \rangle \Rightarrow \langle \sigma[x \mapsto x_{new}], \phi \rangle} \tag{5.7}$$

읽기 명령은 임의의 기호 값을 생성한다 (제약이 없는 신선한 변수).

Rule 5.3.9 (쓰기 (Write))**.**

$$\frac{[\![e]\!]_\sigma = e'}{\langle \sigma, \phi, \texttt{write}(e) \rangle \Rightarrow \langle \sigma, \phi \rangle} \tag{5.8}$$

쓰기 명령은 상태를 변경하지 않지만, 식을 평가한다.

Rule 5.3.10 (단언문 (Assert))**.**

$$\frac{[\![e]\!]_\sigma = b \quad \phi' = \phi \wedge \neg b}{\langle \sigma, \phi, \texttt{assert}(e) \rangle \Rightarrow \langle \sigma, \phi' \rangle} \tag{5.9}$$

단언문은 조건의 부정을 경로 조건에 추가한다. SMT 솔버가 ϕ'가 만족 가능하다고 판단하면, 단언문을 위반하는 입력이 존재함을 의미한다.

5.3.1 식의 기호적 평가

식의 기호적 평가 $[\![e]\!]_\sigma$는 다음과 같이 정의된다:

상수와 변수

$$[\![n]\!]_\sigma = n \quad \text{(정수 상수)} \tag{5.10}$$

$$[\![\texttt{true}]\!]_\sigma = \texttt{true} \quad \text{(불린 상수)} \tag{5.11}$$

$$[\![\texttt{false}]\!]_\sigma = \texttt{false} \quad \text{(불린 상수)} \tag{5.12}$$

$$[\![x]\!]_\sigma = \sigma(x) \quad \text{(변수 참조)} \tag{5.13}$$

단항 연산자

$$[\![\neg e]\!]_\sigma = \neg [\![e]\!]_\sigma \tag{5.14}$$

이항 연산자

$$[\![e_1 + e_2]\!]_\sigma = [\![e_1]\!]_\sigma \oplus_{bv} [\![e_2]\!]_\sigma \tag{5.15}$$

$$[\![e_1 - e_2]\!]_\sigma = [\![e_1]\!]_\sigma \ominus_{bv} [\![e_2]\!]_\sigma \tag{5.16}$$

$$[\![e_1 \times e_2]\!]_\sigma = [\![e_1]\!]_\sigma \otimes_{bv} [\![e_2]\!]_\sigma \tag{5.17}$$

$$[\![e_1 \div e_2]\!]_\sigma = [\![e_1]\!]_\sigma \oslash_{bv} [\![e_2]\!]_\sigma \tag{5.18}$$

$$[\![e_1 \bmod e_2]\!]_\sigma = [\![e_1]\!]_\sigma \mod_{bv} [\![e_2]\!]_\sigma \tag{5.19}$$

$$[\![e_1 < e_2]\!]_\sigma = [\![e_1]\!]_\sigma <_{bv} [\![e_2]\!]_\sigma \tag{5.20}$$

$$[\![e_1 = e_2]\!]_\sigma = [\![e_1]\!]_\sigma =_{bv} [\![e_2]\!]_\sigma \tag{5.21}$$

$$[\![e_1 \wedge e_2]\!]_\sigma = [\![e_1]\!]_\sigma \wedge [\![e_2]\!]_\sigma \tag{5.22}$$

$$[\![e_1 \vee e_2]\!]_\sigma = [\![e_1]\!]_\sigma \vee [\![e_2]\!]_\sigma \tag{5.23}$$

여기서 $\oplus_{bv}, \ominus_{bv}, \otimes_{bv}, \oslash_{bv}, \mod_{bv}, <_{bv}, =_{bv}$는 32비트 비트벡터 연산을 나타낸다.

5.3.2 예제

Example 5.3.11 (조건문의 기호 실행). *다음 프로그램을 고려하자:*

```
y := 10;
if (x > 0) {
    y := y + 1
} else {
    y := y - 1
}
assert ( y == 11 || y == 9 )
```

기호 실행 과정:

1. *초기 상태:* $\sigma_0 = \{x \mapsto x_0, y \mapsto y_0\}$, $\phi_0 =$ `true`
2. $y := 10$*:*
   $y_1 = 10$
   $\sigma_1 = \{x \mapsto x_0, y \mapsto y_1\}$, $\phi_1 = (y_1 = 10)$
3. *Then 분기* ($x_0 > 0$)*:*
   $y_2 = y_1 + 1$
   $\sigma_{then} = \{x \mapsto x_0, y \mapsto y_2\}$, $\phi_{then} = \phi_1 \wedge (x_0 > 0) \wedge (y_2 = y_1 + 1)$
4. *Else 분기* ($x_0 \leq 0$)*:*
   $y_3 = y_1 - 1$
   $\sigma_{else} = \{x \mapsto x_0, y \mapsto y_3\}$, $\phi_{else} = \phi_1 \wedge (x_0 \leq 0) \wedge (y_3 = y_1 - 1)$

5. *병합:*

$y_4 = ite(x_0 > 0, y_2, y_3)$

$\sigma_2 = \{x \mapsto x_0, y \mapsto y_4\},$

$\phi_2 = \phi_{then} \vee \phi_{else}$

최종 제약:

$$\begin{aligned}\phi_2 = (y_1 = 10) \wedge \\ ((x_0 > 0 \wedge y_2 = y_1 + 1) \vee (x_0 \leq 0 \wedge y_3 = y_1 - 1)) \wedge \\ (y_4 = ite(x_0 > 0, y_2, y_3))\end{aligned}$$

```
y = 10

if (x > 0)
    y = y + 1
else
    y = y − 1


assert (y == 11 || y == 8)
```

```
y1 = 10

if (x > 0)
    y2 = y1 + 1
else
    y3 = y1 − 1

y4 = ite (x > 0, y2, y3)

assert (y4 == 11 || y4 == 9)
```

Figure 5.10: 예제 5.3.11를 기호 실행할 때 할당문의 왼쪽 변수 이름을 변경

Example 5.3.12 (Assert 검증)**.** *다음 프로그램의 단언문을 검증하자:*

```
int x;
read(x);
y := x + 1;
assert(y > x);
```

기호 실행:

1. ***read***(x)*:* x_0*는 제약 없는 기호 값*

2. $y := x + 1$*:* $y_0 = x_0 + 1$

3. ***assert***$(y > x)$*:* $\phi = \neg(y_0 > x_0) \equiv (y_0 \leq x_0)$

제약 시스템:

$$\phi = (y_0 = x_0 + 1) \wedge (y_0 \leq x_0)$$

이 제약은 불만족 가능(UNSAT)이므로, 단언문을 위반하는 입력이 존재하지 않는다. 즉, 단언문은 항상 참이다.

Example 5.3.13 (32비트 정수에서 Assert 실패)**.** *다음 프로그램의 단언문을 수학적 정수가 아니라 C와 같은 32비트 부호 있는* `int` *의미(2의 보수, 범위* $[-2^{31}, 2^{31}-1]$*)로 해석하여 검증하자.*

```
int x;
read(x);
y := x + 1;    // 32-bit signed overflow 가능
assert(y > x);
```

기호 실행:

1. `read(x)`*:* x_0*는 제약 없는 기호 값 (32비트* `int`*)*

2. $y := x + 1$*:*

$$y_0 = (x_0 + 1) \bmod 2^{32}$$

여기서 y_0*와* x_0*는 부호 있는 32비트 정수로 해석된다.*

3. `assert(y > x)`*:*

$$\phi = \neg(y_0 > x_0) \equiv (y_0 \le x_0)$$

따라서 단언 위반(Assertion fail)에 해당하는 제약 시스템은

$$\phi = \big(y_0 = (x_0 + 1) \bmod 2^{32}\big) \wedge (y_0 \le x_0)$$

이다.

이제 이 제약이 만족 가능한지 살펴보자. x_0*가 32비트* `int`*의 최댓값인*

$$x_0 = 2^{31} - 1$$

이라고 두면,

$$x_0 + 1 = 2^{31} \equiv -2^{31} \pmod{2^{32}}$$

이므로

$$y_0 = -2^{31}, \quad y_0 \le x_0$$

를 만족한다. 즉, 위 제약 시스템은 SAT이며, $x = 2^{31} - 1$*과 같은 구체 입력에서* `y`*는 오버플로우로 인해 음수가 되고,* `assert(y > x)`*는 실패한다.*

정리하면, 32비트 `int` *의미에서는 단언문을 위반하는 입력이 존재하므로, 이 단언문은 항상 참이 아니다.*

Example 5.3.14 (오버플로우를 고려한 강화된 Assert 검증)**.** *위 프로그램에서 오버플로우를 명시적으로 예외 조건으로 두어, 단언문을 다음과 같이 보강하자.*

```
int x;
read(x);
```

```
y := x + 1;    // 32-bit signed overflow 가능
assert(y > x || x + 1 < x);
```

여기서 `x + 1 < x`는 32비트 부호 있는 비교에서 `x+1`이 `x`보다 작아지는 경우, 즉 `x+1` 계산 중 오버플로우가 발생했다는 상황을 나타낸다.

기호 실행:

1. *`read(x)`: x_0는 제약 없는 기호 값 (32비트 `int`)*
2. *$y := x + 1$:*

$$y_0 = (x_0 + 1) \bmod 2^{32}$$

3. *`assert(y > x || x + 1 < x)`:*

$$\psi = \neg\big(y_0 > x_0 \ \vee \ x_0 + 1 < x_0\big) \equiv (y_0 \le x_0) \wedge (x_0 + 1 \ge x_0)$$

(모든 비교 연산은 32비트 부호 있는 비교이다.)

따라서 단언 위반(Assertion fail)에 해당하는 제약 시스템은

$$\psi = \big(y_0 = (x_0 + 1) \bmod 2^{32}\big) \wedge (y_0 \le x_0) \wedge (x_0 + 1 \ge x_0)$$

이다.

이 제약이 만족 가능한지 살펴보자.

- ***오버플로우가 발생하지 않는 경우:*** *이때는 $y_0 = x_0 + 1$이고, 부호 있는 비교에서 항상 $y_0 > x_0$이다. 따라서 단언식의 첫 번째 분기 $y_0 > x_0$가 참이며, 동시에 $(y_0 \le x_0)$를 만족시킬 수 없으므로 ψ는 거짓이다.*
- ***오버플로우가 발생하는 경우:*** *예를 들어 $x_0 = 2^{31} - 1$이면*

$$y_0 = (x_0 + 1) \bmod 2^{32} = -2^{31}$$

이고, 부호 있는 비교에서 $x_0 + 1 < x_0$가 성립한다. 따라서 단언식의 두 번째 분기 $x_0 + 1 < x_0$가 참이며, 동시에 $(x_0 + 1 \ge x_0)$를 만족시킬 수 없으므로 ψ는 역시 거짓이다.

즉, 32비트 `int` 의미에서

$$y_0 > x_0 \quad \text{또는} \quad x_0 + 1 < x_0$$

둘 중 하나는 항상 참이며, 이 둘이 동시에 거짓인 상태는 존재하지 않는다. 따라서 ψ는 UNSAT이고, 단언문을 위반하는 입력은 존재하지 않는다.

정리하면, 오버플로우를 명시적으로 고려한 강화된 단언문

```
assert(y > x || x + 1 < x);
```

은 32비트 `int` 의미에서도 항상 참이며, 성공적으로 검증된다.

기호 실행 후 수집된 경로 조건 ϕ를 Z3 SMT 솔버에 전달하여 다음과 같이 검증한다.

- **SAT**: 만족 가능 ⇒ 모델(테스트 입력)을 생성
- **UNSAT**: 불만족 가능 ⇒ 해당 경로는 실행 불가능
- **UNKNOWN**: 결정 불가능 (타임아웃 등)

초기 상태에 제약을 부과하고 프로그램을 실행하는 전방향 실행(Forward Execution)과 최종 상태에 제약을 부과하고 역방향으로 프로그램 실행을 거슬러 올라오는 후방향 실행(Backward Execution)이 있다. 후방향 실행은 특정 최종 상태에 도달하기 위한 입력을 찾을 때 유용하다.

5.3.3 한계와 도전 과제

분기문이 n개 있는 프로그램은 최대 2^n개의 경로를 가질 수 있기 때문에 경로 폭발 (Path Explosion) 문제가 있다.

$$\#\text{ paths} \leq 2^{\#\text{ branches}}$$

이 문제를 경로 선택 휴리스틱 (search heuristics), 경로 병합 (path merging), 제약 단순화 (constraint simplification) 방법으로 완화를 시도할 수 있다.

반복문 처리에서 언롤링 기법의 한계가 있다. 고정된 언롤링 횟수 n으로는 모든 경로를 탐색할 수 없고, 적응형 언롤링, 반복 불변식(loop invariant) 사용 등의 고급 기법 필요하기 때문이다.

이론에 따라 NP-complete 또는 더 어렵고, 복잡한 제약으로 SAT도 UNSAT도 아닌 타임아웃이 발생할 수도 있다. 제약 캐싱, 증분적 해결 등으로 최적화하려는 접근 방법이 연구되고 있다.

WHILE 언어에 대한 기호 실행의 형식적 명세를 제시하였다.

1. 기호 상태와 경로 조건의 정의
2. 각 명령에 대한 기호 실행 규칙
3. SSA 형태와 Phi 노드를 통한 분기 병합
4. 반복문 언롤링 기법
5. Z3 SMT 솔버와의 통합

기호 실행은 프로그램 검증, 테스트 생성, 버그 탐지 등 다양한 응용 분야에서 강력한 도구로 사용된다. 경로 폭발 문제 등의 확장성 이슈가 있으나, 다양한 최적화 기법을 통해 실용적인 프로그램 분석이 가능하다.

참고문헌

- Floyd, R. W., "Assigning meanings to programs," Proceedings of Symposia in Applied Mathematics Vol. 19 (1967), pp. 19-32.

- King, James C. "Symbolic execution and program testing." Communications of the ACM 19.7 (1976): 385-394.
- Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." Communications of the ACM 56.2 (2013): 82-90.
- De Moura, Leonardo, and Nikolaj Bjørner. "Z3: An efficient SMT solver." International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2008.
- Jelvis, Tikhon. "Symbolic Execution in Haskell." GitHub Repository: https://github.com/TikhonJelvis/imp

Chapter 6

신경망 검증

신경망 검증 강의 노트는 별도의 PDF를 만들어 병합

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BC93] Paul Bumbulis and Donald D. Cowan. Re2c: A more versatile scanner generator. In *USENIX Winter Conference*, pages 63–74, 1993.

[BL73] David Elliott Bell and Leonard J. LaPadula. Secure computer system: Mathematical foundations and model. Technical Report M74-244, MITRE Corporation, Bedford, MA, 1973.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[Gra88] Robert S. Gray. Fast text scanning with regular expressions. *Technical Report*, 1988.

[Har94] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, 1994.

[Les75] Michael E. Lesk. Lex – a lexical analyzer generator. In *Proceedings of the 1975 Summer Meeting*. Bell Laboratories, 1975.

[MY60] Robert McNaughton and Hisao Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, 1960.

[Pax95] Vern Paxson. *Flex: The Fast Lexical Analyzer*, 1995. Available at https://github.com/westes/flex.

[PS02] François Pottier and Vincent Simonet. Information flow inference for ml. *SIGPLAN Not.*, 37(1):319–330, January 2002.

[SM03] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2–3):167–187, January 1996.

신경망 검증 소개[1]

아우스 알바르구시**(Aws Albarghouthi)**
위스콘신 대학교 매디슨 캠퍼스(University of Wisconsin–Madison)

아랍 글자의 저자 이름: أوس البرغوثي

[1][번역] Aws Albarghouthi, Introduction to Neural Network Verification, University of Wisconsin–Madison. https://verifieddeeplearning.com/

Part I

신경망과 정확성**(Neural Networks & Correctness)**

Chapter 1

새로운 시작

> 자유분방하게 생각을 쏟아내던 시절을 거의 잊은 채, 문장을 다듬는 기교에 지나치게 몰두하고 있었다 (He had become so caught up in building sentences that he had almost forgotten the barbaric days when thinking was like a splash of color landing on a page).
>
> —에드워드 세인트 오빈(Edward St. Aubyn), *모유 (Mother's Milk)*

1.1 튜링으로 거슬러 올라가다(It Starts With Turing)

이 책은 *신경망*이 어떤 바람직한 성질 집합에 따라 동작하는지를 *검증(verifying)*하는 것에 관한 이야기다. 검증과 신경망은 지금까지는 거의 접점이 없었던 전혀 다른 연구 분야였지만, 최근에 들어서야 두 영역을 잇는 다리가 놓이기 시작했다. 흥미롭게도, 두 분야 모두 그 기원을 앨런 튜링의 짧고도 비극적인 생애 중 2년에 불과한 기간에서 찾을 수 있다.

1949년, 튜링은 *Checking a Large Routine* (Alan, 1949)이라는 거의 알려지지 않은 논문을 발표했다. 이는 매우 선구적인 작품이었다. 여기서 튜링은 우리가 작성하는 프로그램이 정말로 의도한 대로 동작한다는 것을 어떻게 증명할 수 있을까라는 질문을

윌리암 피네건(William Finnegan)의 *바바리안 데이즈(Barbarian Days)*에서 인용.

던졌다. 그리고 나서 팩토리얼 함수를 계산하는 프로그램의 정확성을 실제로 증명해 보였다. 구체적으로, 튜링은 그의 작은 코드 조각이 항상 종료하며 입력의 팩토리얼을 항상 산출한다는 것을 증명했다. 그 증명은 우아했다. 프로그램을 단일 명령어 단위로 쪼개고, 각 명령어마다 보조정리를 증명한 다음, 그 보조정리들을 이어붙여 전체 프로그램의 정확성을 입증한 것이다. 오늘날까지도 프로그램 검증은 1949년 튜링의 증명 방식에 크게 의존한다. 그리고, 이 책에서 보게 되겠지만, 신경망의 증명 역시 마찬가지다.

튜링이 팩토리얼 프로그램의 정확성을 증명하기 바로 1년 전인 1948년에, 그는 아마도 더 멀리 내다본 논문 *Intelligent Machinery*를 집필했다.[1] 이 논문에서 튜링은 *무조직 기계(unorganized machines)*를 제안했다. 그는 이 기계들이 유아의 인간 대뇌피질을 모방한다고 주장했고, 오늘날 우리가 유전 알고리즘이라고 부르는 방식을 통해 학습할 수 있음을 보였다. 무조직 기계는 오늘날 우리가 신경망이라고 부르는 것의 매우 단순한 형태였다.

1.2 딥러닝의 출현과 부상(The Rise of Deep Learning)

신경망 훈련에 관한 연구는 튜링의 1948년 논문 이후 계속 이어졌다. 그러나 지난 10여 년 동안 폭발적으로 인기를 얻은 것은 알고리즘적 통찰, 하드웨어 발전, 그리고 학습을 위한 방대한 데이터가 한꺼번에 쏟아져 들어온 덕분이다.

현대의 신경망은 *심층(deep)* 신경망이라 불리며, 이를 학습하는 방법을 *딥러닝(deep learning)*이라고 한다. 딥러닝은 특히 컴퓨터 비전과 자연어 처리와 같은 복잡한 계산 과제에서 놀라운 성과 향상을 가능하게 했다. 예를 들어, 이미지 속의 사물과 사람을 인식하거나 언어를 번역하는 일 등이다. 오늘날 연구 공동체는 음악 생성에서 수학 정리 증명에 이르기까지 더 어려운 문제에 딥러닝을 적용하는 방법을 매일같이 탐구하고 있다.

딥러닝의 발전은 소프트웨어가 무엇인지, 무엇을 할 수 있는지, 어떻게 만들어지는지에 대한 우리의 인식을 바꾸어 놓았다. 현대의 소프트웨어는 점점 전통적인 수작업 코드와 자동으로 학습된—때로는 끊임없이 학습하는—신경망의 혼합체가 되어가고 있다. 하지만 심층 신경망은 취약하고 예상치 못한 결과를 낼 수 있다. 자율주행차와 같은 민감한 환경에서 점점 더 많이 사용되기 때문에, 우리는 이 시스템들을 검증하고 그 동작에 대한 형식적 보장을 제공하는 것이 필수적이다. 다행히 수십 년간 쌓아온 프로

[1] *Intelligent Machinery*는 Turing (1969)에 재수록되어 있다.

그램 검증 연구가 있으므로, 이를 토대로 삼을 수 있다. 그렇다면, 우리는 정확히 무엇을 검증해야 할까?

1.3 신경망에 우리가 기대하는 것(What do We Expect of Neural Networks?)

튜링이 팩토리얼 프로그램의 정확성을 증명할 때, 그는 컴퓨터가 수학 연산을 수행하도록 프로그래밍될 것이지만 그 연산이 잘못될 수 있음을 걱정했다. 그래서 자신의 팩토리얼 구현이 수학적 정의와 동일하다는 것을 증명했다. 이러한 개념을 *기능적 정확성(functional correctness)*이라 하며, 이는 프로그램이 어떤 수학적 함수를 충실히 구현한다는 뜻이다. 기능적 정확성은 많은 환경에서 매우 중요하다. 예를 들어, 암호학적 기본 연산이나 항공기 제어기의 버그는 재앙을 초래할 수 있다.

그러나 딥러닝의 세계에서 기능적 정확성을 증명하는 것은 비현실적이다. 이미지에서 고양이를 올바르게 인식하거나 영어를 힌디어로 올바르게 번역한다는 것이 무슨 의미인가? 우리는 이러한 작업을 수학적으로 정의할 수 없다. 사실, 번역이나 이미지 인식 같은 과제를 딥러닝에 맡기는 이유 자체가 우리가 그 과제를 수학적으로 명확히 규정할 수 없기 때문이다.

그렇다면 어떻게 할 것인가? 신경망 검증은 불가능한 것일까? 아니다! 신경망이 무엇을 해야 하는지 정확히 규정할 수는 없더라도, 종종 그 바람직하거나 바람직하지 않은 성질을 특성화할 수 있다. 몇 가지 예를 살펴보자.

강건성(Robustness)

신경망의 가장 많이 연구된 정확성 속성은 *강건성(robustness)*이다. 이는 일반적 성질이며, 심층 학습 모델이 취약하다는 사실이 잘 알려져 있기 때문이다 (Szegedy et al., 2014). 강건성이란 입력에 작은 섭동이 있어도 출력이 바뀌지 않아야 한다는 뜻이다. 예를 들어, 사진 속 몇 개의 픽셀만 바꿨는데 신경망이 나를 사람 대신 찬장이라고 인식해서는 안 된다. 또는 강의 녹음에 사람이 들을 수 없는 잡음을 추가했는데 신경망이 이를 15세기 명나라 역사 강의로 인식해서는 안 된다. 우스꽝스러운 예를 떠나, 강건성이 결여되면 안전과 보안에 심각한 위험이 될 수 있다. 예를 들어, 카메라로 교통 표지를 인식하는 자율주행차의 경우, 스톱 사인에 약간의 낙서만 해도 차량이 이를 인식하지 못해 사고로 이어질 수 있음이 밝혀졌다 (Eykholt et al., 2018). 또는 악성코드 탐지

신경망의 경우를 생각해 보자. 악성코드의 이진 파일을 약간만 수정했는데 탐지기가 갑자기 이를 설치해도 안전하다고 판단해서는 안 된다.

안전성(Safety)

안전성(safety)은 프로그램이 *나쁜 상태(bad state)*에 도달하지 않아야 한다는 폭넓은 성질이다. 여기서 "나쁜 상태"의 정의는 과제에 따라 달라진다. 예를 들어, 공장에서 신경망으로 제어되는 로봇의 경우, 인부를 위험에 빠뜨리지 않도록 일정 속도 제한을 넘지 않거나 위험한 구역으로 들어가지 않도록 해야 한다. 또 다른 연구된 예로, 항공기 충돌 회피 시스템을 구현하는 신경망이 있다 (Katz et al., 2017). 관심 있는 성질 중 하나는, 만약 다른 항공기가 왼쪽에서 접근한다면 신경망은 반드시 우회하도록 결정을 내려야 한다는 것이다.

일관성(Consistency)

신경망은 이미지와 같은 예제를 통해 세상에 대해 학습한다. 그 결과 때로는 물리 법칙이나 현실적인 시나리오에 대한 기본 공리를 놓칠 수 있다. 예를 들어, 이미지 속 객체와 그 관계를 인식하는 신경망이 객체 A가 B 위에 있고, B가 C 위에 있으며, C가 다시 A 위에 있다고 말할 수 있다. 하지만 이는 불가능하다! (적어도 우리가 아는 세상에서는.)

또 다른 예를 들어, 축구 경기장을 카메라로 추적하는 신경망을 생각해 보자. 어떤 한 프레임에서는 호날두가 오른쪽에 있다고 했다가, 다음 프레임에서는 왼쪽에 있다고 말해서는 안 된다. 호날두가 빠르긴 하지만, 최근 시즌에는 예전만큼 빠르지 않다.

앞으로(Looking Ahead)

나는 신경망의 성질을 검증하는 일이 얼마나 중요한지 충분히 설득했기를 바란다. 다음 두 장에서는 신경망이 정확히 어떤 모습인지를 형식적으로 정의하고 (힌트: 매우 못생긴 프로그램이다), 이어서 신경망의 정확성 속성을 형식적으로 명세하는 언어를 만들 것이다. 이를 통해 이러한 성질을 입증할 수 있는 검증 알고리즘의 토대를 마련할 것이다.

Chapter 2

Neural Networks as Graphs

딥러닝이 무엇이고 무엇이 아닌지에 대한 엄밀한 정의는 없다. 사실, 이 글을 쓰는 시점에도 인공지능 커뮤니티에서는 명확한 정의를 두고 뜨거운 논쟁이 벌어지고 있다. 이 장에서는 신경망을 실수(real number)에 대한 연산들로 이루어진 그래프로 일반적으로 정의한다. 실제로 이러한 그래프의 형태(이를 *아키텍처*라고 부른다)는 임의적이지 않다: 연구자와 실무자들은 다양한 과제에 맞게 새로운 아키텍처를 세심하게 설계한다. 예를 들어, 이 글을 쓰는 시점에서 이미지 인식을 위한 신경망은 자연어 과제를 위한 신경망과 보통 다르게 생겼다.

우선 우리는 비형식적으로 그래프를 소개하고 몇 가지 인기 있는 아키텍처를 살펴볼 것이다. 그런 다음 그래프와 그 의미론을 형식적으로 정의할 것이다.

2.1 The Neural Building Blocks

신경망은 각 노드가 연산을 수행하는 그래프이다. 전체적으로, 이 그래프는 실수 벡터에서 실수 벡터로 가는 함수, 즉 $\mathbb{R}^n \to \mathbb{R}^m$ 꼴의 함수를 나타낸다. 다음의 아주 간단한 그래프를 생각해 보자.

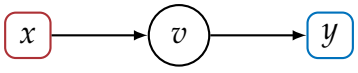


Figure 2.1 아주 간단한 신경망

빨간 노드는 *입력* 노드로, 실수 x를 노드 v에 전달한다. 노드 v는 x에 어떤 연산을 수행하고 그 값을 *출력* 노드 y로 내보낸다. 예를 들어, v는 단순히 $2x + 1$을 반환할 수도

있고, 이를 함수 $f_v : \mathbb{R} \to \mathbb{R}$로 나타내면:

$$f_v(x) = 2x + 1$$

우리의 모델에서 출력 노드 역시 어떤 연산을 수행할 수 있다. 예를 들어,

$$f_y(x) = \max(0, x)$$

이 둘을 합치면, 이 단순한 그래프는 다음과 같은 함수 $f : \mathbb{R} \to \mathbb{R}$를 인코딩한다:

$$f(x) = f_y(f_v(x)) = \max(0, 2x + 1)$$

Transformations and Activations

위 예에서 함수 f_v는 *아핀(affine)* 함수이다: 즉, 입력을 상수 값으로 곱한 뒤(여기서는 $2x$), 상수를 더한다(여기서는 1). 함수 f_y는 *활성화(activation)* 함수인데, 이는 입력에 따라 *켜지거나 꺼지기* 때문이다. 입력이 음수이면 f_y는 0을 출력(꺼짐)하고, 그렇지 않으면 입력값을 그대로 출력(켜짐)한다. 구체적으로, Figure 2.2에 나타낸 f_y는 *정류 선형 단위*(ReLU)라 불리며, 현대 심층 신경망에서 매우 널리 쓰이는 활성화 함수이다 (Nair and Hinton, 2010). 활성화 함수는 신경망에 비선형성을 추가하는 역할을 한다.

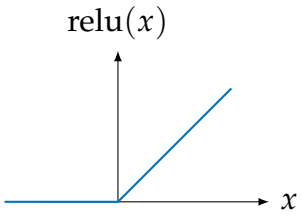


Figure 2.2 정류 선형 단위(ReLU)

또 다른 인기 있는 활성화 함수로는 sigmoid가 있다:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

이는 출력이 0과 1 사이로 제한된다. Figure 2.3 참고.

문헌과 실무에서는 아핀 함수와 활성화 함수를 하나의 연산으로 묶어서 다루는 경우가 많다. 우리의 그래프 모델은 이를 포착할 수 있지만, 이후 장에서 그래프를 분석할 때 단순화를 위해 보통 두 연산을 별도의 노드로 나누어 표현한다.

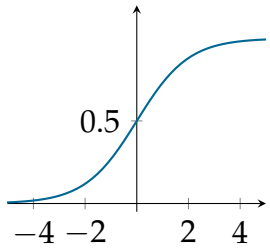


Figure 2.3 시그모이드 함수

Universal Approximation

이러한 활성화 함수가 특별한 이유는 무엇일까? 간단히 말하면, 실제로 잘 동작한다는 것이다. 즉, 복잡한 과제를 학습할 수 있는 신경망을 만들어 준다. 흥미로운 점은 ReLU나 sigmoid와 아핀 함수를 사용한 신경망으로 임의의 연속 함수를 근사할 수 있다는 것이다. 이를 *보편 근사 정리(universal approximation theorem)*라고 하며 (Hornik et al., 1989), 사실 이 결과는 ReLU와 sigmoid보다 훨씬 더 일반적이다. 거의 모든 활성화 함수가 동작하는데, 다항식이 아니기만 하면 된다 (Leshno et al., 1993)! 보편 근사의 대화형 예시는 Nielsen (2018, Ch.4)을 추천한다.

2.2 Layers and Layers and Layers

일반적으로 신경망은 노드와 화살표가 여기저기 얽힌 복잡한 그래프일 수 있다. 그러나 실제로는 보통 *층(layered)* 구조를 가진다. Figure 2.4을 보자. 여기에는 입력이 3개, 출

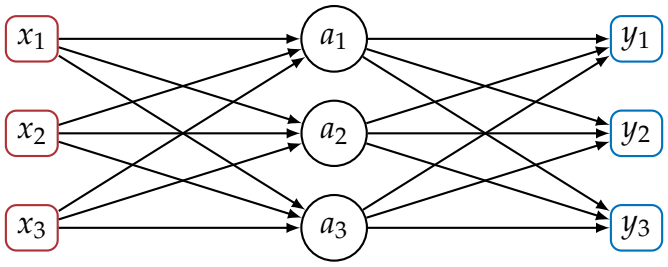


Figure 2.4 다층 퍼셉트론

력이 3개 있으며, 이는 $\mathbb{R}^3 \to \mathbb{R}^3$ 꼴의 함수를 의미한다. 노드들이 층을 이루고 있음을

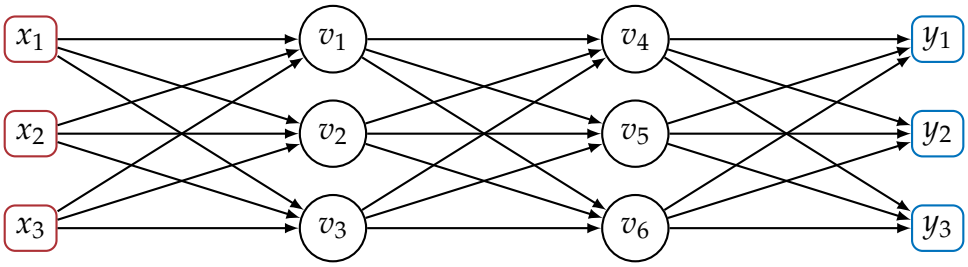


Figure 2.5 두 개의 은닉층을 가진 다층 퍼셉트론

알 수 있다: 입력층, 출력층, 그리고 그 사이에 있는 *은닉층(hidden layer)*이다. 이런 형태의 그래프—즉 아키텍처—는 거창하게도 *다층 퍼셉트론(multilayer perceptron,* MLP*)*이라 불린다. 보통 MLP 에는 여러 은닉층이 있다. Figure 2.5는 두 개의 은닉층을 가진 MLP 이다. MLP 의 층은 *완전 연결(fully connected)* 층이라 불리는데, 각 노드가 이전 층의 모든 출력으로부터 입력을 받기 때문이다.

신경망은 보통 *분류기(classifier)*로 사용된다: 즉, 입력(예: 이미지의 픽셀)을 받아 그 이미지가 무엇인지(이미지의 클래스)를 예측한다. 분류를 할 때, MLP 의 출력층은 각 클래스의 확률을 나타낸다. 예를 들어, y_1은 입력이 의자일 확률, y_2는 TV일 확률, y_3은 소파일 확률이다. 확률이 0과 1 사이이고 합이 1이 되도록 하기 위해, 마지막 층은 *소프트맥스(softmax)* 함수를 사용한다. 소프트맥스는 일반적으로 다음과 같이 정의된다. 클래스 개수가 n일 때, 출력 노드 y_i에 대해:

$$f_{y_i}(x_1, \ldots, x_n) = \frac{\exp(x_i)}{\sum_{k=1}^{n} \exp(x_k)}$$

왜 이것이 잘 작동할까? 예를 들어 클래스가 2개, 즉 $n = 2$라 하자. 먼저,

$$f_{y_1}, f_{y_2} \in [0, 1]$$

임을 확인할 수 있다. 이는 분자와 분모가 모두 양수이고, 분자가 분모보다 크지 않기 때문이다. 또한,

$$f_{y_1}(x_1, x_2) + f_{y_2}(x_1, x_2) = 1$$

임을 알 수 있다:

$$f_{y_1}(x_1, x_2) + f_{y_2}(x_1, x_2) = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} + \frac{e^{x_2}}{e^{x_1} + e^{x_2}} = 1$$

이 두 사실을 합치면 확률 분포가 됨을 알 수 있다. 소프트맥스의 대화형 시각화는 Nielsen (2018, Chapter 3)의 온라인 책을 참고하라.

신경망의 출력 $(y_1, \ldots, y_n)$이 주어졌을 때, 우리는

$$\mathsf{class}(y_1, \ldots, y_n)$$

을 가장 큰 원소의 인덱스(동점은 없다고 가정)로 정의한다. 즉, 가장 큰 확률을 가진 클래스이다. 예를 들어, $\mathsf{class}(0.8, 0.2) = 1$, $\mathsf{class}(0.3, 0.7) = 2$이다.

2.3 Convolutional Layers

신경망에서 또 다른 종류의 층은 *합성곱(convolutional)* 층이다. 이 층은 컴퓨터 비전 작업에서 널리 사용되지만, 자연어 처리에서도 쓰인다. 직관은 이렇다: 이미지를 볼 때, 패턴을 찾기 위해 스캔하고 싶다. 합성곱 층은 이를 가능하게 해준다: 즉, *커널(kernel)*이라 불리는 연산을 정의하여 이미지의 모든 픽셀 영역이나 문장의 모든 단어 구간에 적용한다. 예를 들어, 4단어 문장에서 각 단어가 입력을 정의하는 입력층 크기가 4라고 하자 (Figure 2.6 참고). 여기에는 커널 $\{v_1, v_2, v_3\}$가 있으며, 이는 연속된 두 단어 $(x_1, x_2), (x_2, x_3), (x_3, x_4)$에 적용된다. 이 커널은 $\mathbb{R}^2$ 입력을 받으므로 크기가 2라 한다. 이 커널은 1차원인데, 입력이 실수 벡터이기 때문이다. 실제로는 2차원 이상의 커널을 사용한다. 예를 들어, 픽셀 하나가 실수인 흑백 이미지를 스캔할 때, $\mathbb{R}^{10\times 10} \to \mathbb{R}$ 꼴의 커널을 사용하여 입력의 모든 10×10 부분 이미지에 적용할 수 있다.

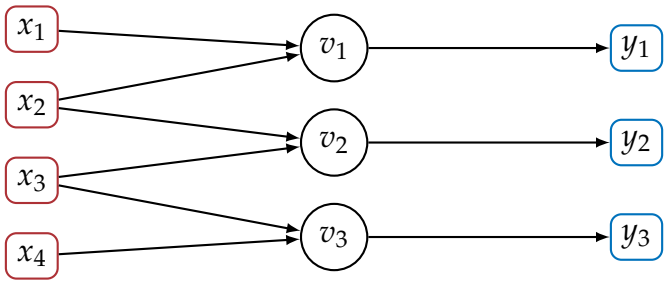


Figure 2.6 1차원 합성곱

일반적으로, *합성곱 신경망(convolutional neural network,* CNN*)*은 입력에 여러 커널을 적용하고—여러 층을 쌓아—각 커널의 정보를 모은다(*풀링(pooling)*). 이러한 연산들은 이후 장에서 이러한 네트워크의 성질을 검증할 때 다시 만나게 될 것이다.[1]

2.4 Where are the Loops?

지금까지 살펴본 신경망은 연속된 여러 개의 수학적 함수들을 합성한 것처럼 보인다. 그렇다면 반복문은 어디에 있을까? 신경망에 반복문을 넣을 수 있을까? 실제로 신경망 그래프는 *역사이클* 유향 그래프(DAG)이다. 이 덕분에 *역전파*(backpropagation) 알고리즘을 사용해 신경망을 학습시킬 수 있다.

그렇다 해도, 반복문이 있는 것처럼 보이는 인기 있는 신경망 부류가 있다. 다만 그 반복 횟수가 입력의 크기에 의해 정해진다는 점에서 단순하다. *순환 신경망*(recurrent neural network, RNN)이 그 전형적인 예로, 텍스트와 같은 시퀀스 데이터에 주로 사용된다. RNN 의 그래프는 보통 노드 v에 자기 루프가 달린 아래 그림처럼 그려진다.

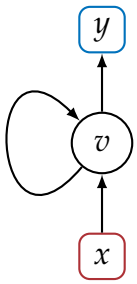


Figure 2.7 순환 신경망

사실상 이 그래프는 해당 루프를 유한 번 풀어(unroll) 늘어놓은 무한 가족의 역사이클 그래프를 나타낸다. 예컨대 Figure 2.8은 길이 3으로 언롤한 것이다. 이는 3개의 입력을 받아 3개의 출력을 내는 역사이클 그래프임을 알 수 있다. 아이디어는 이렇다. 길이가 n인 문장을 받으면, RNN 을 길이 n으로 언롤하여 그 문장에 적용한다.

[1] 실제로 CNN 을 구성할 때는 여러 매개변수가 있다. 예: 커널 개수, 커널이 적용되는 입력의 크기, 커널의 보폭(stride) 등. 그러나 이 책에서는 관심 대상이 아니다. 우리는 주로 신경망의 핵심 구성 요소에 집중하는데, 이것들이 검증상의 도전을 결정하기 때문이다.

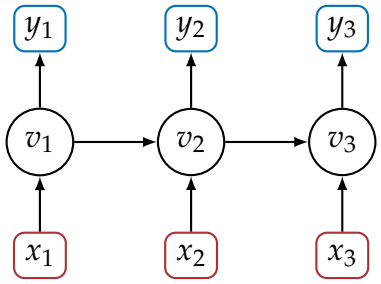


Figure 2.8 언롤된 순환 신경망

프로그래밍 관점에서 생각해 보면, 입력이 주어지면 네트워크를 실제로 실행하지 않고도—즉 정적으로—필요한 반복 횟수를 쉽게 결정할 수 있다. 이는, 예컨대 반복 횟수가 입력의 복잡한 함수로 결정되어 실제 실행해 보아야만 반복 횟수를 알 수 있는 일반 프로그램과 대조적이다. 이제부터는 신경망을 역사이클 그래프로 형식화할 것이다.

2.5 Structure and Semantics of Neural Networks

예쁜 그림 구경은 여기까지. 이제 예쁜 기호를 보자. 이제부터 신경망을 유향 역사이클 그래프로 형식적으로 정의하고 그 성질을 논의한다.

Neural Networks as DAGs

신경망은 유향 역사이클 그래프 $G = (V, E)$이며, 다음이 주어진다.

- V는 유한한 노드 집합,
- $E \subseteq V \times V$는 간선 집합,
- $V^{\text{in}} \subset V$는 공집합이 아닌 입력 노드 집합,
- $V^{\text{o}} \subset V$는 공집합이 아닌 출력 노드 집합, 그리고
- 각 비입력 노드 v에는 함수 $f_v : \mathbb{R}^{n_v} \to \mathbb{R}$가 연관되며, 여기서 n_v는 v를 목표로 하는 간선의 개수이다. v가 입력으로 받는 실수 벡터 $\mathbb{R}^{n_v}$는 $(v', v) \in E$인 모든 노드 v'의 출력들로 이루어진다. 단순화를 위해(그리고 일반성을 잃지 않고) 노드 v가 실수 하나만 출력한다고 가정한다.

그래프 G에 매달린 노드가 없고 의미가 명확하도록 하기 위해, 다음의 구조적 성질을 가정한다:

- 모든 노드는 어떤 입력 노드로부터 유향 간선을 따라 도달 가능하다.
- 모든 노드는 어떤 출력 노드에 도달할 수 있다.
- 간선 집합 E와 노드 집합 V에는 각각 고정된 전순서가 존재한다.

$\boldsymbol{x} \in \mathbb{R}^n$은 n항(행) 벡터를 나타내며, 스칼라들의 튜플 $(x_1, \ldots, x_n)$로 표기한다. 여기서 x_i는 $\boldsymbol{x}$의 i번째 성분이다.

Semantics of DAGs

신경망 $G = (V, E)$는 $\mathbb{R}^n \to \mathbb{R}^m$ 꼴의 함수를 정의하며,

$$n = |V^{\mathsf{in}}| \quad \text{and} \quad m = |V^{\mathsf{o}}|$$

이다. 즉, G는 입력 노드들의 값을 출력 노드들의 값으로 사상한다.

구체적으로, 모든 비입력 노드 $v \in V$에 대해, 그 노드가 산출하는 $\mathbb{R}$의 값을 다음과 같이 귀납적으로 정의한다. v를 목표로 하는 모든 간선들의 (가정한 순서에 따른) 순서열을 $(v_1, v), \ldots, (v_{n_v}, v)$라 하자. 그러면 노드 v의 출력은

$$\mathsf{out}(v) = f_v(x_1, \ldots, x_{n_v})$$

이며 여기서 $x_i = \mathsf{out}(v_i)$, $i \in \{1, \ldots, n_v\}$이다.

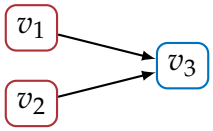
위 정의(의 out)의 기저 사례는 입력 노드들이다. 입력 노드에는 입사 간선이 없기 때문이다. 입력 벡터 $\boldsymbol{x} \in \mathbb{R}^n$가 주어졌다고 하자. 모든 입력 노드의 (가정한 순서에 따른) 순서열을 $v_1, \ldots, v_n$라 하면,

$$\mathsf{out}(v_i) = x_i$$

로 둔다.

A Simple Example

예제 그래프 G를 보자:



여기서 $V^{\text{in}} = \{v_1, v_2\}$, $V^{\text{o}} = \{v_3\}$이다. 이제

$$f_{v_3}(x_1, x_2) = x_1 + x_2$$

라 하고, 입력 벡터가 $(11, 79)$라고 하자. 즉, 노드 v_1은 11, v_2는 79를 받는다. 그러면

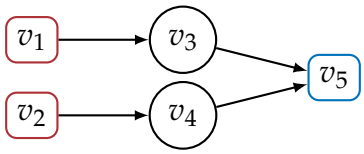
$$\begin{aligned}
\mathsf{out}(v_1) &= 11 \\
\mathsf{out}(v_2) &= 79 \\
\mathsf{out}(v_3) &= f_{v_3}(\mathsf{out}(v_1), \mathsf{out}(v_2)) = 11 + 79 = 90
\end{aligned}$$

가 된다.

Data Flow and Control Flow

우리가 정의한 그래프는 컴파일러와 프로그램 분석 분야에서 *데이터 플로(data-flow)* 그래프로 알려져 있다; 이는 *제어 플로(control-flow)* 그래프와 대조적이다.[2] 제어 플로 그래프는 연산을 수행해야 하는 *순서*, 즉 누가 CPU 의 *제어권*을 가지는지의 흐름을 규정한다. 반면 데이터 플로 그래프는 어떤 노드가 계산을 수행하려면 어떤 데이터가 필요한지를 말해 줄 뿐, 계산의 순서를 강제하지는 않는다. 작은 예로 보는 것이 가장 좋다.

다음 그래프를 생각해 보자.



이 그래프를 명령형 프로그램처럼 보면, $\leftarrow$를 대입 기호로 하여 다음과 같이 나타낼 수 있다.

$$\begin{aligned}
\mathsf{out}(v_3) &\leftarrow f_{v_3}(\mathsf{out}(v_1)) \\
\mathsf{out}(v_4) &\leftarrow f_{v_4}(\mathsf{out}(v_2)) \\
\mathsf{out}(v_5) &\leftarrow f_{v_5}(\mathsf{out}(v_3), \mathsf{out}(v_4))
\end{aligned}$$

[2] TensorFlow 같은 딥러닝 프레임워크에서는 데이터 플로 그래프를 *계산 그래프(computation graph)*라고 부른다.

이 프로그램은 노드 v_3의 출력이 v_4의 출력보다 먼저 계산됨을 규정한다. 하지만 이는 필수가 아니다. v_3의 출력은 v_4의 출력에 의존하지 않기 때문이다. 따라서 같은 그래프의 동등한 구현은 처음 두 연산을 바꿔 쓸 수 있다:

$$\begin{aligned}\mathsf{out}(v_4) &\leftarrow f_{v_4}(\mathsf{out}(v_2))\\ \mathsf{out}(v_3) &\leftarrow f_{v_3}(\mathsf{out}(v_1))\\ \mathsf{out}(v_5) &\leftarrow f_{v_5}(\mathsf{out}(v_3), \mathsf{out}(v_4))\end{aligned}$$

형식적으로, 우리는 그래프 노드들의 임의의 *위상 정렬(topological ordering)*에 따라 $\mathsf{out}(\cdot)$ 값을 계산할 수 있다. 이렇게 하면 어떤 노드의 연산을 수행하기 전에 그 노드의 모든 입력이 계산되었음을 보장한다.

Properties of Functions

지금까지는 노드 v가 실수들에 대한 임의의 함수 f_v를 구현할 수 있다고 가정했다. 실제로 신경망을 효율적으로 학습시키려면, 이러한 함수들은 *미분 가능*하거나 거의 모든 점에서 미분 가능(*almost everywhere*)해야 한다. 앞서 Figure 2.3에서 본 시그모이드는 미분 가능하다. 반면 Figure 2.2의 ReLU는 $x = 0$에서 꺾임이 있어 기울기가 정의되지 않으므로, 거의 모든 점에서만 미분 가능하다.

우리가 관심 가질 함수들 중 많은 수가 *선형* 또는 *구간별 선형(piecewise linear)*이다. 형식적으로, $f : \mathbb{R}^n \to \mathbb{R}$가 다음과 같이 정의될 수 있으면 선형이다:

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} c_i x_i + b$$

여기서 $c_i, b \in \mathbb{R}$이다. 함수가 다음 꼴로 쓸 수 있으면 구간별 선형이다:

$$f(\boldsymbol{x}) = \begin{cases} \sum_i c_i^1 x_i + b^1, & \boldsymbol{x} \in S_1 \\ \vdots & \\ \sum_i c_i^m x_i + b^m, & \boldsymbol{x} \in S_m \end{cases}$$

여기서 S_i들은 서로소인 $\mathbb{R}^n$의 부분집합들이고 $\cup_i S_i = \mathbb{R}^n$이다. 예컨대 ReLU는 다음과 같은 꼴의 구간별 선형 함수이다:

$$\mathrm{relu}(x) = \begin{cases} 0, & x < 0 \\ x, & x \geqslant 0 \end{cases}$$

우리가 나중에 활용할 또 다른 중요한 성질은 *단조성(monotonicity)*이다. 함수 $f : \mathbb{R} \to \mathbb{R}$가 단조 증가라 함은, 임의의 $x \geqslant y$에 대해 $f(x) \geqslant f(y)$가 성립하는 경우를 말한다. 이 장에서 본 두 활성화 함수, ReLU와 시그모이드는 단조 증가 함수이다. 이는 Figures 2.2 and 2.3에서 확인할 수 있다. x가 증가함에 따라 함수값이 결코 감소하지 않는다.

Looking Ahead

이제 신경망을 형식적으로 정의했으니, 그 동작에 대해 질문을 던질 준비가 되었다. 다음 장에서는 그러한 질문들을 형식적으로 명세하는 언어를 정의한다. 그리고 뒤이은 장들에서는 그 질문들에 답하는 알고리즘들을 살펴볼 것이다.

문헌에서 신경망을 논의할 때는 선형대수의 언어를 많이 사용한다—예컨대 Goodfellow et al. (2016)의 종합서를 보라. 선형대수는 한 층의 많은 노드 연산을 이전 층의 출력에 적용되는 하나의 행렬 A로 간결하게 표현할 수 있게 해 준다. 또한 실제로는 행렬 곱셈의 빠르고 병렬적인 구현을 사용하여 신경망을 계산한다. 여기서는 각 노드가 $\mathbb{R}^n \to \mathbb{R}$ 꼴의 함수라는 낮은 수준의 표현을 택한다. 이 관점은 비표준적이지만, 다양한 검증 기법의 제시를 더 깔끔하게 만들어 준다. 문제를 개별 노드와 관련된 더 작은 문제들로 분해할 수 있기 때문이다.

우리가 제시한 신경망의 그래프는 Tensorflow (Abadi et al., 2016)와 PyTorch (Paszke et al., 2019) 같은 딥러닝 프레임워크의 계산 그래프를 더 낮은 수준으로 나타낸 것이다.

신경망은 *미분 가능 프로그램*(differentiable programs)이라 불리는 일반적 프로그램 부류의 한 예이다. 이름에서 알 수 있듯, 미분 가능 프로그램은 도함수를 계산할 수 있는 프로그램으로, 신경망을 학습하는 표준 기법들에 필요한 성질이다. 최근에는 프로그램이 미분 가능하다는 것이 무엇을 의미하는지에 대한 흥미로운 연구들이 있었다 (Abadi and Plotkin, 2020; Sherman et al., 2021). 가까운 미래에는 임의의 미분 가능 프로그램을 사용해 신경망을 정의하고 학습하는 일이 일반화될 가능성이 크다. 오늘날에는 아직 그렇지 않으며, 대부분의 신경망은 몇 가지 보편적인 아키텍처와 연산을 갖는다.

Chapter 3

Correctness Properties

이 장에서는 신경망의 성질을 명세하기 위한 *언어*를 만들 것이다. 명세 언어는 신경망(때로는 여러 신경망)의 동작에 대해 진술을 하는 공식적 방식이다. 이 장에서 우리의 관심사는 오직 성질을 *명세*하는 일이며, 그것을 자동으로 *검증*하는 일은 아니다. 따라서 우리는 복잡한 성질, 터무니없는 성질, 쓸모없는 성질도 자유롭게 명세해 볼 것이다. 책의 뒷부분에서는 특정 검증 알고리즘에 맞게 관심 있는 성질을 제약할 것이다—지금은 재미있게 시작해 보자.

3.1 Properties, Informally

신경망은 함수 $f : \mathbb{R}^n \to \mathbb{R}^m$을 정의한다는 점을 기억하자. 여기서 다룰 성질은 다음과 같은 꼴이다:

> **임의의 입력** x에 대해, 신경망이 **다음과 같은 출력을 낸다** ...

즉, 성질은 네트워크의 입력–출력 행태를 규정할 뿐, 내부가 *어떻게* 답을 만들어 내는지는 규정하지 않는다.

때로는 성질이 더 복잡해서 여러 입력(어쩌면 여러 네트워크)에 대해 이야기할 수 있다:

> **임의의 입력** x, y**에 대해,** ... **만족된다면** ... 신경망들이 **다음과 같은 출력을 낸다** ...

이러한 성질의 앞부분, 즉 입력에 대해 말하는 부분을 *전제(사전조건, precondition)*라 하고; 뒷부분, 출력에 대해 말하는 부분을 *사후조건(postcondition)*이라 한다. 아래에서는 예시를 통해 비형식적으로 성질을 소개한다.

Figure 3.1 왼쪽: mnist 데이터셋의 손글씨 7. 가운데: 동일한 숫자의 밝기를 높인 버전. 오른쪽: 왼쪽 위에 점을 추가한 버전.

Image Recognition

이미지를 입력받아 *개*, *얼룩말* 등 라벨을 예측하는 신경망 f가 있다고 하자. 이러한 분류기에서 중요하게 보장하고 싶은 성질은 *강건성(robustness)*이다. 분류기의 예측이 입력의 작은 변화(섭동)에 의해 바뀌지 않는다면 강건하다고 한다. 예를 들어, 밝기를 조금 바꾸거나 몇몇 픽셀이 손상되어도 분류가 바뀌어서는 안 된다.

f가 *개*로 분류한 어떤 이미지 $\boldsymbol{c}$를 고정하자. $\boldsymbol{c}$가 신경망을 속이도록 고안된 *적대적(adversarial)* 이미지가 아님을 확인하기 위해, 우리는 다음 성질을 *검사*—*증명* 또는 *검증*—할 것이다:

> 임의의 이미지 $\boldsymbol{x}$가 $\boldsymbol{c}$보다 약간 더 밝거나 어둡다면, $f(\boldsymbol{x})$는 *개*를 예측한다

여기서 전제는 $\boldsymbol{c}$보다 더 밝거나 어두운 이미지들의 집합을 규정하고, 사후조건은 f의 분류가 변하지 않음을 규정한다.

강건성은 바람직한 성질이다: 밝기 슬라이더를 조금 움직였다고 분류가 바뀌면 곤란하다. 하지만 우리가 바라는 성질은 이밖에도 많다—대비 변화, 회전, 인스타그램 필터, 화이트 밸런스 등등. 이 지점이 명세 문제의 핵심이다: 우리가 바라는 모든 것을 명세할 수는 없기에, *무엇을* 선택할지 결정해야 한다. (이 부분은 뒤에서 더 다룬다.)

구체적 예시는 Figure 3.1을 보라. MNIST 데이터셋 (LeCun et al., 2010)은 손글씨 숫자 인식을 위한 표준 데이터셋이다. 그림은 손글씨 7과 두 가지 수정 버전을 보여 준다. 하나는 밝기를 높였고, 다른 하나는 왼쪽 위에 가짜 점(잉크가 튄 것처럼)을 추가했다. 신경망이 이 세 이미지를 모두 7로 분류하길 원한다.

Natural-Language Processing

이번에는 f가 영어 문장을 입력받아 긍정/부정을 판별한다고 하자. 이는 온라인 후기나 트윗을 자동 분석할 때 등장하는 문제다. 이 설정에서도 강건성이 중요하다. 예를 들어, 긍정 감성의 문장 $\boldsymbol{c}$가 고정되어 있을 때 다음 성질을 명세할 수 있다:

> 문장 $\boldsymbol{x}$가 $\boldsymbol{c}$에서 몇 가지 철자 오류만 추가된 것이라면, $f(\boldsymbol{x})$는 여전히 긍정을 예측해야 한다

또 다른 예로, 철자 오류 대신 동의어 치환을 상상해 보자:

> 문장 $\boldsymbol{x}$가 $\boldsymbol{c}$에서 일부 단어를 동의어로 바꾼 것이라면, $f(\boldsymbol{x})$는 여전히 긍정을 예측해야 한다

예컨대 다음 두 영화 리뷰는 모두 긍정으로 분류되어야 한다:

> *This movie is delightful*
> *This movie is enjoyable*

위 두 성질을 결합해, 동의어 치환이나 철자 오류가 있어도 예측이 바뀌지 않는다는 더 강한 성질을 기술할 수 있다.

Source Code

신경망 f가 악성코드 분류기여서, 코드 조각을 입력받아 악성 여부를 판정한다고 하자. 공격자는 악성코드를 살짝 수정해 신경망을 속이고 정상 프로그램으로 오인시키려 할 수 있다. 공격 기법 중 하나는, 동작은 바꾸지 않되 신경망을 속이도록 코드를 덧붙이는 것이다. 이를 다음과 같이 명세할 수 있다: 악성코드 $\boldsymbol{c}$가 하나 주어졌을 때, 다음 성질을 기술하자:

> 프로그램 $\boldsymbol{x}$가 $\boldsymbol{c}$와 동등하고 문법적으로도 유사하다면, $f(\boldsymbol{x})$는 악성으로 예측해야 한다

Controllers

지금까지의 예는 모두 강건성에 관한 것이었다. 조금 다른 성질을 보자. 로봇의 행동을 결정하는 컨트롤러가 있다고 하자. 컨트롤러는 세계의 상태를 보고 왼쪽/오른쪽/앞/뒤로 움직일지 결정한다. 당연히 우리는 로봇이 벽이나 사람, 다른 로봇 같은 장애물로 움직이지 않기를 원한다. 따라서 다음 성질을 명세할 수 있다:

> 임의의 상태 $\boldsymbol{x}$에 대해, 로봇의 오른쪽에 장애물이 있으면 $f(x)$는 *오른쪽*을 예측해서는 안 된다

이와 유사한 성질을 각 방향마다 하나씩 기술할 수 있다.

3.2 A Specification Language

우리의 명세는 다음과 같은 모습이 될 것이다:

$$\begin{gathered}\{\ \textit{precondition}\ \} \\ \boldsymbol{r} \leftarrow f(\boldsymbol{x}) \\ \{\ \textit{postcondition}\ \}\end{gathered}$$

*전제(사전조건)*는 true/false로 평가되는 불리언 함수(*술어*)이다. 전제는 우리가 논의하는 신경망에 입력으로 쓰일 변수들에 대해 정의된다. 그 변수들을 $\boldsymbol{x}_i$로 표기한다. 명세의 가운데 부분은 신경망이 정의하는 함수 호출들의 나열이다; 위 예에서는 f를 한 번 호출하고 반환값을 변수 r에 담았다. 일반적으로는 다음처럼 여러 대입문의 순서를 허용한다:

$$\begin{gathered}\{\ \textit{precondition}\ \} \\ \boldsymbol{r_1} \leftarrow f(\boldsymbol{x_1}) \\ \boldsymbol{r_2} \leftarrow g(\boldsymbol{x_2}) \\ \vdots \\ \{\ \textit{postcondition}\ \}\end{gathered}$$

마지막으로, 사후조건은 전제에 등장한 변수들 $\boldsymbol{x}_i$와 대입으로 생성된 변수들 $\boldsymbol{r}_j$에 대한 불리언 술어이다.

명세를 비형식적으로 읽는 법은 이렇다:

> 전제를 참으로 만드는 임의의 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$에 대해, $\boldsymbol{r}_1 = f(\boldsymbol{x_1}), \boldsymbol{r}_2 = g(\boldsymbol{x}_2), \ldots$ 라 하자. 그러면 사후조건이 참이다.

만약 올바름 성질이 참이 아니라면, 즉 사후조건이 거짓이라면, 그 성질이 *성립하지 않는다(does not hold)*고 말한다.

Example 3.A 앞 절의 이미지 밝기 예시를 떠올리자. $\boldsymbol{c}$를 실제의 그레이스케일 이미지라고 하자. 각 원소는 픽셀의 세기(0은 검정, 1은 흰색)를 나타낸다. 예를 들어 Figure 3.1

의 MNIST 예시에서는 각 숫자가 784 픽셀(28×28)로 표현되며, 각 픽셀은 0과 1 사이의 수이다. 그러면 다음 명세를 기술할 수 있다. 이는 비형식적으로 $\boldsymbol{c}$의 밝기를 바꾸더라도 분류가 변하지 않아야 함을 뜻한다 (Section 2.2의 class 정의를 상기하라):

$$\begin{gathered}\{\ |\boldsymbol{x} - \boldsymbol{c}| \leqslant \mathbf{0.1}\ \} \\ \boldsymbol{r}_1 \leftarrow f(\boldsymbol{x}) \\ \boldsymbol{r}_2 \leftarrow f(\boldsymbol{c}) \\ \{\ \mathsf{class}(\boldsymbol{r}_1) = \mathsf{class}(\boldsymbol{r}_2)\ \}\end{gathered}$$

이 명세를 차례로 살펴보자:

전제 이미지 $\boldsymbol{x}$를 취하되, 각 픽셀이 $\boldsymbol{c}$의 대응 픽셀과 최대 0.1만큼만 차이나는 경우로 제한한다. 여기서 $\boldsymbol{x}$와 $\boldsymbol{c}$는 같은 크기라 가정하고, $\leqslant$는 성분별(pointwise)로 정의한다.[1]

대입 $\boldsymbol{r}_1$을 $f(\boldsymbol{x})$의 결과, $\boldsymbol{r}_2$를 $f(\boldsymbol{c})$의 결과로 둔다.

사후조건 그러면 벡터 $\boldsymbol{r}_1$과 $\boldsymbol{r}_2$의 예측 라벨이 동일하다. 분류 설정에서 벡터 $\boldsymbol{r}_i$의 각 성분은 특정 라벨의 확률을 의미함을 상기하자. class는 벡터의 최댓값 성분의 인덱스를 추출하는 약기호이다.

■

Counterexamples

어떤 성질에 대한 *반례(counterexample)*란 전제의 변수들(즉 $\boldsymbol{x}_i$들)에 대한 값 배정으로서 사후조건을 거짓으로 만드는 것이다. Example 3.A에서 반례는 이미지 $\boldsymbol{x}$인데, f의 분류가 이미지 $\boldsymbol{c}$의 분류와 다르면서 $\boldsymbol{c}$와의 거리 $|\boldsymbol{x} - \boldsymbol{c}|$가 0.1보다 작은 경우이다.

Example 3.B 다음은 구체적 예시다(이미지 인식이 아니라 입력에 1을 더하는 단순 함수):

$$\begin{gathered}\{\ x \leqslant 0.1\ \} \\ r \leftarrow x + 1 \\ \{\ r \leqslant 1\ \}\end{gathered}$$

[1] 성분별 연산 $|\cdot|$는 ℓ_∞ 노름으로 알려져 있으며, **??**에서 다른 노름들과 함께 형식적으로 다룬다.

이 성질은 성립하지 않는다. x를 0.1로 두어 보자. 그러면 $r \leftarrow 1 + 0.1 = 1.1$이 된다. 따라서 사후조건이 거짓이 된다. 즉, $x = 0.1$이 반례이다. ■

A Note on Hoare Logic

우리의 명세 언어는 *호어 논리(Hoare logic)* (Hoare, 1969)의 명세와 유사하다. 호어 논리에서의 명세는 세 부분으로 이루어진 *호어 삼중항(Hoare triple)*이라 부른다. 우리의 명세도 마찬가지다. 호어 논리에는 그러한 명세의 타당성을 증명할 수 있는 추론 규칙들이 갖추어져 있다. 이 책에서는 호어 논리의 규칙들을 따로 정의하지는 않지만, 책 전반에 걸쳐 많은 규칙이 암묵적으로 등장할 것이다.

3.3 More Examples of Properties

이제 여러 가지 예시 성질을 우리 명세 언어로 적어 보자.

Equivalence of Neural Networks

이미지 인식용 신경망 f가 있고 이를 새로운 신경망 g로 교체하고자 한다고 하자. 아마 g가 더 작고 빠르기 때문일 것이다. 들어오는 이미지 흐름(stream)에서 실행할 계획이라면 효율이 매우 중요하다. 검증하고 싶은 것 중 하나는 f와 g의 *동등성*이다. 이 성질은 다음처럼 쓸 수 있다:

$$\begin{gathered}\{\ \mathsf{true}\ \} \\ \boldsymbol{r_1} \leftarrow f(\boldsymbol{x}) \\ \boldsymbol{r_2} \leftarrow g(\boldsymbol{x}) \\ \{\ \mathsf{class}(\boldsymbol{r}_1) = \mathsf{class}(\boldsymbol{r}_2)\ \}\end{gathered}$$

전제가 true임에 주목하라. 이는 임의의 이미지 $\boldsymbol{x}$에 대해 f와 g의 예측 라벨이 같기를 원한다는 뜻이다. true 전제는 신경망의 입력(여기서는 $\boldsymbol{x}$)이 *무제약*임을 뜻한다. 이 명세는 매우 강하다: 모든 가능한 입력에 대해 f와 g가 동일 분류를 내야만 성립하는데, 실제로는 그럴 가능성이 매우 낮다.

대안으로, f와 g가 어떤 이미지 부분집합에 대해(밝기를 약간 올리거나 내리는 정도를 포함해) 같은 예측을 낸다고 기술할 수 있다. S를 유한한 이미지 집합이라 하자.

그러면:

$$\{\ \boldsymbol{x}_1 \in S,\ |\boldsymbol{x}_1 - \boldsymbol{x}_3| \leqslant \mathbf{0.1},\ |\boldsymbol{x}_1 - \boldsymbol{x}_2| \leqslant \mathbf{0.1}\ \}$$
$$\boldsymbol{r}_1 \leftarrow f(\boldsymbol{x}_2)$$
$$\boldsymbol{r}_2 \leftarrow g(\boldsymbol{x}_3)$$
$$\{\ \mathsf{class}(\boldsymbol{r}_1) = \mathsf{class}(\boldsymbol{r}_2)\ \}$$

이는 다음을 뜻한다: 이미지 $\boldsymbol{x}_1$을 하나 고르고, 그 밝기가 약간 다른 두 변형 $\boldsymbol{x}_2$와 $\boldsymbol{x}_3$를 만든다. 그러면 f와 g는 두 이미지에 대해 같은 분류를 내야 한다.

이는 첫 시도보다 실용적인 동등성 개념이다. 첫 시도는 모든 가능한 입력에 대해 f와 g의 일치를 강제했지만, 대부분의 이미지(픽셀 조합)는 무의미한 잡음이고 우리는 그 분류엔 관심이 없다. 반면 위 명세는 S에 있는 이미지들과 *닮은* 무한 집합으로 동등성을 제한한다.

Collision Avoidance

다음 예시는 Katz et al. (2017)의 기념비적 연구 이후 검증 문헌에서 널리 다루어진 사례다. 여기에는 자율 항공기에 탑재된 충돌 회피 시스템이 있다. 시스템은 침입 항공기를 탐지하고 무엇을 할지 결정한다. 신경망을 사용하는 이유는 복잡성 때문이다: 학습된 신경망은 매우 거대한 규칙표보다 훨씬 작다. 어떤 의미에서 신경망은 규칙을 *압축*해 효율적으로 실행 가능한 프로그램으로 만든다.

신경망의 입력은 다음과 같다:

- v_{own}: 우리 항공기의 속도
- v_{int}: 침입 항공기의 속도
- a_{int}: 현재 비행 방향에 대한 침입 항공기의 각도
- a_{own}: 침입기에 대한 우리 항공기의 각도
- d: 두 항공기 사이의 거리
- $prev$: 직전 행동

위 값들을 입력받아 신경망은 조종 방향을 결정한다: 좌/우, 강한 좌/우, 혹은 아무것도 하지 않기. 구체적으로, 신경망은 가능한 각 행동에 점수를 매기고, *가장 낮은 점수*의 행동을 택한다.

상상할 수 있듯, 여기서는 많은 것이 잘못될 수 있고, 잘못되면—재앙이다! Katz et al. (2017)는 여러 성질을 선정해 검증한다. 이 성질들이 모든 상황을 포괄하지는 않지만, 중요한 점검 항목이다. 그중 하나를 보자. 침입 항공기가 멀리 있다면 *아무것도 하지 않기*의 점수가 어떤 임계값보다 낮아야 한다는 성질이다.

$$\{\ d \geqslant 55947,\ v_{own} \geqslant 1145,\ v_{int} \leqslant 60\ \}$$
$$\boldsymbol{r} \leftarrow f(d, v_{own}, v_{int}, \ldots)$$
$$\{\ \text{벡터 } \boldsymbol{r}\text{에서 아무것도 하지기의 점수가 1500 미만}\ \}$$

전제는 두 항공기 사이의 거리가 55947피트보다 크고, 우리 항공기의 속도가 높으며, 침입 항공기의 속도가 낮음을 규정한다. 사후조건은 아무것도 하지기를 선택하는 점수가 낮아야 함(임계값 미만)을 규정한다. 직관적으로, 두 항공기가 매우 멀고 속도도 크게 다르면, 괜히 당황할 필요가 없다.

Katz et al. (2017)는 이와 같은 여러 성질을 탐구하고, 충돌 회피 맥락의 강건성 성질도 고려한다. 하지만 이렇게 구체적 성질을 어떻게 도출할까? 간단하지 않다. 이 경우에는 충돌 회피 시스템을 잘 아는 도메인 전문가가 필요하고, 그럼에도 모든 코너 케이스를 포괄하지 못할 수 있다. 검증 커뮤니티(저자 포함)의 여러 연구자들은 명세가 검증보다 더 어렵다고 주장한다—즉, 어려운 부분은 *올바른 질문을 던지는 일*이다!

Physics Modeling

다음 예시는 Qin et al. (2019)에 따른 것이다. 우리는 신경망이 진자의 운동 같은 물리 법칙을 내재화하길 원한다. 임의의 시점에서 진자의 상태는 (v, h, w)의 삼중쌍으로, 수직 위치 v, 수평 위치 h, 각속도 w이다. 진자의 상태가 주어지면, 시간이 이산 단계로 나뉜다고 가정할 때, 다음 시점의 상태를 예측한다.

자연스러운 점검 항목은 신경망이 이해한 진자의 운동이 에너지 보존 법칙을 따르는지다. 임의의 시점에서 진자의 에너지는 위치 에너지와 운동 에너지의 합이다. (고등학교 물리 기억나는가?) 진자가 올라갈수록 위치 에너지는 증가하고 운동 에너지는 감소한다; 내려갈수록 반대다. 둘의 합(총 에너지)은 시간에 따라 *감소만* 해야 한다. 이를 다음처럼 기술할 수 있다:

$$\{\ \text{true}\ \}$$
$$v', h', w' \leftarrow f(v, h, w)$$
$$\{\ E(h', w') \leqslant E(h, w)\ \}$$

여기서 $E(h, w)$는 진자의 에너지로, 질량 m, 중력가속도 g일 때 위치 에너지 mgh와, 길이 l일 때 운동 에너지 $0.5ml^2w^2$의 합이다.

Natural-Language Processing

장 앞부분의 자연어 예시를 다시 보자. 문장을 긍/부정으로 분류하는데, 단어를 동의어로 바꿔도 분류가 변하지 않길 원했다. 이를 우리의 언어로 표현해 보자: $\boldsymbol{c}$를 길이 n의 고정 문장이라 하자. 벡터 $\boldsymbol{c}$의 각 원소는 단어의 *임베딩*이라 불리는 실수라 가정한다. 또한 시소러스 T가 있어, 단어를 주면 동의어 집합을 돌려준다고 하자.

$$\begin{gathered}\{\ 1 \leqslant i \leqslant n,\ w \in T(c_i),\ \boldsymbol{x} = \boldsymbol{c}[i \mapsto w]\ \} \\ \boldsymbol{r}_1 \leftarrow f(\boldsymbol{x}) \\ \boldsymbol{r}_2 \leftarrow f(\boldsymbol{c}) \\ \{\ \mathsf{class}(\boldsymbol{r}_1) = \mathsf{class}(\boldsymbol{r}_2)\ \}\end{gathered}$$

전제는 변수 $\boldsymbol{x}$가 문장 $\boldsymbol{c}$와 같되, i번째 성분만 시소러스에서 가져온 단어 w로 치환되었음을 명시한다. 기호 $\boldsymbol{c}[i \mapsto w]$는 $\boldsymbol{c}$의 i번째 성분을 w로 바꾼 것을 뜻하고, c_i는 $\boldsymbol{c}$의 i번째 성분을 뜻한다.

위 성질은 한 단어를 동의어로 바꾸는 경우만 허용한다. 두 단어로 확장하면 다음과 같다(보기 흉하지만, 동작은 한다):

$$\begin{gathered}\{\ 1 \leqslant i,j \leqslant n\ ,\ i \neq j\ ,\ w_i \in T(c_i)\ ,\ w_j \in T(c_j)\ ,\ \boldsymbol{x} = \boldsymbol{c}[i \mapsto w_i, j \mapsto w_j]\ \} \\ \boldsymbol{r}_1 \leftarrow f(\boldsymbol{x}) \\ \boldsymbol{r}_2 \leftarrow f(\boldsymbol{c}) \\ \{\ \mathsf{class}(\boldsymbol{r}_1) = \mathsf{class}(\boldsymbol{r}_2)\ \}\end{gathered}$$

Monotonicity

신경망에서 바랄 수 있는 표준 수학적 성질로 *단조성(monotonicity)* (Sivaraman et al., 2020)이 있다. 더 큰 입력이 더 큰 출력을 이끌어야 한다는 뜻이다. 예컨대 주택 가격을 예측하는 웹서비스를 상상해 보자. 모델은 면적에 대해 단조적이어야 한다—면적을 늘렸는데 가격이 줄어들어서는 안 되며, 어쩌면 증가해야 한다. 또 수술 중 합병증 위험을 추정하는 모델을 상상하자. 환자의 나이를 늘렸는데 위험이 줄어들어서는 안 된다. (의사는 아니지만, 예시로는 좋다.) 이를 우리의 언어로는 다음처럼 쓸 수 있다:

$$\begin{gathered}\{\ \boldsymbol{x} > \boldsymbol{x}'\ \} \\ \boldsymbol{r} \leftarrow f(\boldsymbol{x}) \\ \boldsymbol{r}' \leftarrow f(\boldsymbol{x}') \\ \{\ \boldsymbol{r}' \geqslant \boldsymbol{r}'\ \}\end{gathered}$$

즉, $\boldsymbol{x} > \boldsymbol{x}'$인 임의의 입력 쌍을 택했을 때, $f(\boldsymbol{x}) \geqslant f(\boldsymbol{x}')$이길 원한다. 물론 문제 도메인에 따라 사후조건을 *강화*해 *엄격 부등식*으로 만들 수도 있다.

Looking Ahead

책의 첫 부분을 마쳤다. 우리는 신경망을 정의했고, 그 성질을 어떻게 명세할지 살펴보았다. 이제부터는 성질을 자동으로 검증하는 다양한 방법을 논의할 것이다.

강건성 문제, 특히 이미지 인식 분야에서는 엄청난 양의 연구가 진행되었다. 강건성 부족은 Szegedy et al. (2014)에서 처음 관찰되었고, 그 이후로 강건성 위반(적대적 예제)을 탐지하고 방어하는 수많은 접근법이 제안되었다. 이들은 뒤에서 정리하겠다. 우리가 정의한 자연어 처리용 강건성 성질은 Ebrahimi et al. (2018)과 Huang et al. (2019)을 따른다.

Part II

Constraint-Based Verification

Chapter 4

Logics and Satisfiability

이 부분에서는 제약 기반 검증 기법을 살펴본다. 아이디어는 올바름 성질(correctness property)을 제약식 집합으로 인코딩하는 것이다. 그 제약을 풀어 보면 올바름 성질이 성립하는지 여부를 결정할 수 있다.

우리가 사용할 제약은 *일차 논리*(FOL)의 공식들이다. FOL 은 매우 크고 아름다운 세계이지만, 신경망은 그중 아주 작고 아늑한 한 구석—바로 이 장에서 탐구할 구석—에만 산다.

4.1 Propositional Logic

가장 순수한 것, *명제 논리*부터 시작하자. 명제 논리의 공식 F는 불리언 변수(전통적으로 $p, q, r, \ldots$라 부른다) 위에서 다음 문법으로 정의된다:

$$
\begin{array}{rll}
F := & \mathsf{true} & \\
 & \mathsf{false} & \\
 & \mathit{var} & \text{변수} \\
 & \mid F \wedge F & \text{논리곱 (and)} \\
 & \mid F \vee F & \text{논리합 (or)} \\
 & \mid \neg F & \text{부정 (not)}
\end{array}
$$

본질적으로, 명제 논리의 공식은 불리언 변수와 AND 게이트($\wedge$), OR 게이트($\vee$), NOT 게이트($\neg$)로 이루어진 회로를 정의한다. 부정의 연산자 우선순위가 가장 높고, 그다음이 논리곱, 마지막이 논리합이다. 결국 모든 프로그램은 회로로 정의될 수 있다. 컴퓨터에서 모든 것은 비트이고, 메모리는 유한하며, 따라서 변수의 수도 유한하기 때문이다.

$fv(F)$를 공식에 등장하는 *자유* 변수들의 집합을 나타내는 기호로 쓰겠다. 여기서는 단순히 공식에 문법적으로 등장하는 모든 변수를 뜻한다.

Example 4.A 예를 들어, 다음 공식을 보자

$$F \triangleq (p \wedge q) \vee \neg r$$

여기서 $\triangleq$ 표기의 용도에 주의하라. 이는 F를 오른쪽의 공식으로 *문법적으로 정의*한다는 뜻이지, 두 공식이 *의미론적으로 동치*라는 뜻은 아니다(곧 다룬다). 이때 F의 자유 변수 집합은 $fv(F) = \{p, q, r\}$이다. ■

Interpretations

F를 변수 집합 $fv(F)$ 위의 공식이라 하자. F의 해석(interpretation) I는 변수 집합 $fv(F)$에서 true 또는 false로 가는 사상이다. 해석 I와 공식 F가 주어졌을 때, $I(F)$는 F에서 $fv(F)$의 각 변수를 I가 주는 값으로 치환한 공식을 뜻한다.

Example 4.B 다음 공식이 있다고 하자

$$F \triangleq (p \wedge q) \vee \neg r$$

그리고 해석이

$$I = \{p \mapsto \mathsf{true},\ q \mapsto \mathsf{true},\ r \mapsto \mathsf{false}\}$$

라고 하자. I는 변수와 그 해석의 쌍들을 모은 집합으로 표기한다. I를 F에 적용하면

$$I(F) \triangleq (\mathsf{true} \wedge \mathsf{true}) \vee \neg\mathsf{false}$$

를 얻는다. ■

Evaluation Rules

공식을 평가(혹은 단순화)하기 위한 다음 규칙들을 정의한다. $\equiv$ 오른쪽의 공식은 왼쪽 공식과 *동치*이지만, 문법적으로 더 단순하다:

$$\begin{array}{lcll} \mathsf{true} \wedge F & \equiv & F & \text{논리곱} \\ F \wedge \mathsf{true} & \equiv & F & \\ \mathsf{false} \wedge F & \equiv & \mathsf{false} & \\ F \wedge \mathsf{false} & \equiv & \mathsf{false} & \end{array}$$

$$
\begin{array}{lcll}
\mathsf{false} \vee F & \equiv & F & \text{논리합} \\
F \vee \mathsf{false} & \equiv & F & \\
\mathsf{true} \vee F & \equiv & \mathsf{true} & \\
F \vee \mathsf{true} & \equiv & \mathsf{true} & \\
 & & & \\
\neg\mathsf{true} & \equiv & \mathsf{false} & \text{부정} \\
\neg\mathsf{false} & \equiv & \mathsf{true} &
\end{array}
$$

자유 변수가 없는 공식이라면, 위 규칙들을 반복 적용하여 결국 true 또는 false를 얻게 된다. 위 규칙들을 반복 적용하여 얻을 수 있는 F의 가장 단순한 형태를 $\mathsf{eval}(F)$로 표기한다.

Satisfiability

공식 F가 *만족 가능*(SAT)하다는 것은 어떤 해석 I가 존재하여

$$\mathsf{eval}(I(F)) = \mathsf{true}$$

가 되는 것을 말한다. 이때 I를 F의 *모델*이라 하며

$$I \models F$$

라고 표기한다. 또한 $I \not\models F$는 I가 F의 모델이 아님을 뜻한다. 위 정의로부터 $I \not\models F$ iff $I \models \neg F$가 성립한다.

동치적으로, 공식 F가 *만족 불가능*(UNSAT)하다는 것은 모든 해석 I에 대해 $\mathsf{eval}(I(F)) = \mathsf{false}$인 것을 말한다.

Example 4.C 공식 $F \triangleq (p \vee q) \wedge (\neg p \vee r)$를 보자. 이 공식은 만족 가능하다. 한 모델은 $I = \{p \mapsto \mathsf{true}, q \mapsto \mathsf{false}, r \mapsto \mathsf{true}\}$이다. ■

Example 4.D 공식 $F \triangleq (p \vee q) \wedge \neg p \wedge \neg q$를 보자. 이 공식은 만족 불가능하다. ■

Validity and Equivalence

신경망의 성질을 증명할 때 우리는 종종 *타당성(validity)*을 묻게 된다. 공식 F가 타당하다는 것은 가능한 모든 해석 I가 F의 모델이라는 뜻이다. 따라서 공식 F는 $\neg F$가 만족불가능일 때에 한하여(iff) 타당하다.

Example 4.E 다음 공식은 타당하다: $F \triangleq (\neg p \vee q) \vee p$. 임의의 해석 I를 골라도 $I \models F$임을 확인할 수 있다. ■

두 공식 A, B에 대해, 모든 모델 I가 A의 모델이면 B의 모델이고, 그 역도 성립하면 두 공식이 *동치*라고 한다. 동치는 $A \equiv B$로 표기한다. 공식을 다룰 때 유용한 동치들이 많다. 임의의 공식 A, B, C에 대해 논리곱과 논리합의 교환법칙이 성립한다:

$$\begin{aligned} A \wedge B &\equiv B \wedge A \\ A \vee B &\equiv B \vee A \end{aligned}$$

부정을 안쪽으로 밀어 넣을 수 있다:

$$\begin{aligned} \neg(A \wedge B) &\equiv \neg A \vee \neg B \\ \neg(A \vee B) &\equiv \neg A \wedge \neg B \end{aligned}$$

더 나아가, 논리합에 대한 논리곱의 분배법칙과 그 역(일명 *드모르간 법칙*)이 성립한다:

$$\begin{aligned} A \vee (B \wedge C) &\equiv (A \vee B) \wedge (A \vee C) \\ A \wedge (B \vee C) &\equiv (A \wedge B) \vee (A \wedge C) \end{aligned}$$

Implication and Bi-implication

우리는 자주 *함의* $A \Rightarrow B$를 공식

$$\neg A \vee B$$

로 표기해서 쓴다. 마찬가지로 *쌍방 함의* $A \Leftrightarrow B$는

$$(A \Rightarrow B) \wedge (B \Rightarrow A)$$

로 표기한다.

4.2 Arithmetic Theories

이제 *이론(theory)*을 사용해 명제 논리를 확장할 수 있다. 각 불리언 변수를, 다양한 타입의 변수들 위에서 정의된 더 복잡한 불리언 표현으로 바꾼다. 예를 들어, *선형 실수 산술*(LRA) 이론을 사용할 수 있는데, 이때 불리언 표현은 다음과 같은 꼴이 된다:

$$x + 3y + z \leqslant 10$$

또는 *배열* 이론을 사용할 수도 있어, 표현이

$$a[10] = x$$

처럼 생길 수 있다. 여기서 a는 정수로 인덱싱되는 배열이다. 이 밖에도 *비트벡터*(기계 산술 모델링)나 *문자열*(문자열 조작 모델링) 등 많은 이론들이 연구되었다. 이제 만족가능성 문제는 *이론에 대한 만족가능성*(SMT)이라 불리며, 특정 이론의 해석에 비추어 만족가능성을 판단한다.

이 절에서는 LRA 이론에 집중할 것이다. 그 이유는 (1) 결정 가능(decidable)하고 (2) 다음 장에서 보겠지만 신경망 연산의 큰 부분을 표현할 수 있기 때문이다.

Linear Real Arithmetic

LRA 에서는 각 명제 변수를 다음 꼴의 선형 부등식으로 대체한다:

$$\sum_{i=1}^{n} c_i x_i + b \leqslant 0$$

또는

$$\sum_{i=1}^{n} c_i x_i + b < 0$$

여기서 $c_i, b \in \mathbb{R}$이고 $\{x_i\}_i$는 고정된 변수 집합이다. 예를 들어, 다음과 같은 공식을 가질 수 있다:

$$(x + y \leqslant 0 \ \wedge \ x - 2y < 10) \vee x > 100$$

$>$와 $\geqslant$는 $<$와 $\leqslant$로 다시 쓸 수 있음을 주의하라. 또한 계수 c_i가 0이면 항 $c_i x_i$는 생략한다. 위의 $x > 100$ 부등식에는 변수 y가 등장하지 않는 것처럼 말이다. 등식 $x = 0$은 $x \geqslant 0 \wedge x \leqslant 0$의 합성으로 쓸 수 있다. 마찬가지로 $x \neq 0$은 $x < 0 \vee x > 0$으로 쓸 수 있다.

Models in LRA

명제 논리와 마찬가지로, LRA 의 공식 F에서 자유 변수 집합 $fv(F)$는 공식에 등장하는 변수들의 집합이다.

공식 F의 해석 I는 모든 자유 변수에 실수를 할당하는 것이다. 해석 I가 F의 모델, 즉 $I \models F$이려면 $\mathsf{eval}(I(F)) = \mathsf{true}$여야 한다. 여기서 LRA 공식에 대한 단순화 규칙의 확장은 직관적이다: 산술 부등식 평가에 대한 표준 규칙만 더해 주면 된다. 예컨대 $2 \leqslant 0 \equiv \mathsf{false}$ 같은 것들이다.

Example 4.F 다음 공식을 보자:

$$F \triangleq x - y > 0 \ \wedge \ x \geqslant 0$$

이에 대한 한 모델은

$$\{x \mapsto 1, y \mapsto 0\}$$

이다. I를 F에 적용하면, 즉 $I(F)$는

$$1 - 0 > 0 \ \wedge \ 1 \geqslant 0$$

가 되고, 평가 규칙을 적용하면 true를 얻는다. ■

Real vs. Rational

문헌에서는 LRA 를 *선형 유리수 산술*로 부르는 경우도 있다. 그 이유는 서로 연관된 두 가지다: 첫째, 실제로 우리가 공식을 작성할 때 상수들은 유리수이다—예컨대 π를 컴퓨터 메모리에 정확히 표현할 수는 없다. 둘째, F에 등장하는 계수가 모두 유리수라고 하자. 그렇다면 F가 만족 가능할 경우, 모든 자유 변수에 유리수를 할당하는 모델이 항상 존재한다.

Example 4.G 간단한 공식 $x < 10$을 생각해 보자. $\{x \mapsto \pi\}$도 $x < 10$의 모델이지만, $\{x \mapsto 1/2\}$처럼 x를 유리수로 두는 만족 해도 존재한다. 이는 항상 성립한다: 공식 자체에 무리수가 포함되어 있지 않는 한(예: $x = \pi$), 무리수 값만을 갖는 모델만 존재하도록 공식을 만들 수 없다. ■

Non-Linear Arithmetic

LRA 공식의 만족가능성 판정은 NP-완전 문제다. 이론을 다항식 부등식을 허용하도록 확장하면, 알려진 최선의 알고리즘은 최악의 경우 공식 크기에 대해 이중지수 시간이 든다 (Caviness and Johnson, 2012). 초월함수—exp, cos, log 등—를 허용하면 만족가능성은 *결정 불가능*해진다 (Tarski, 1998). 따라서 실용적 목적상 우리는 LRA 에 머무른다. 비록 NP-완전(이 용어는 이론가들의 등골을 오싹하게 한다)이지만, 실제로는 큰 공식에도 확장 가능한 매우 효율적인 알고리즘들이 존재한다.

Connections to MILP

LRA 의 공식과 LRA 에 대한 SMT 문제는 *혼합 정수 선형 계획법*(MILP) 문제와 동치이다. 많은 SMT 솔버가 있듯, MILP 솔버도 많다. 따라서 "그냥 MILP 솔버를 쓰면 안 되나?" 라는 질문이 자연스럽다. 요약하면, 쓸 수 있고, 때로는 SMT 솔버보다 실제로 더 빠를 수도 있다. 그러나 SMT 프레임워크는 매우 일반적이고 유연하다. LRA 로만 공식을 쓸 수 있는 것이 아니라, (1) 다양한 이론의 공식도 쓸 수 있고, (2) 이론들을 *결합*한 공식도 쓸 수 있다.

첫째, 실제로 신경망은 실수나 유리수 산술 위에서 동작하지 않는다. 부동소수점, 고정소수점, 또는 정수 기계 산술로 동작한다. 신경망 분석을 가능한 한 정밀하게 하고 싶다면, 연산을 비트 수준으로 인코딩하고 SMT 솔버가 사용하는 비트벡터 이론을 쓸 수 있다. (놀랍게도 기계 산술은 선형 실수 산술보다 현실적으로 풀기 더 비싼 경우가 많다. 그래서 대개 신경망은 실수 산술로 인코딩한다.)

둘째, 앞으로 신경망이 어디에나 등장함에 따라, 신경망을 다른 코드 조각들과 *함께* 검증해야 할 것이다. 예를 들어, 텍스트를 파싱하여 신경망이 소비할 형식으로 바꿔 주는 코드가 있다고 하자. 이런 코드를 분석하려면 문자열 연결 등 연산을 다루는 *문자열* 이론이 필요할 수 있다. SMT 솔버는 이론 *결합*을 위한 정리-증명 기법을 사용하므로, 문자열과 선형 산술을 동시에 다루는 공식을 쓸 수 있다.

이런 이유로, 이 책에서는 제약 기반 검증의 목표 솔버로 SMT 를 사용한다: 다양한 일차 이론을 제공하고, 그것들을 결합할 수 있게 해 준다. 다만 이 글을 쓰는 시점에, 제약 기반 검증 연구의 대부분은 선형 실수 산술 인코딩에 초점을 맞추고 있음을 덧붙여 둔다.

Looking Ahead

다음 장에서는 신경망의 의미론과 올바름 성질을 LRA 공식으로 인코딩하는 법을 살펴보고, 이를 통해 SMT 솔버로 자동 검증이 가능해짐을 보일 것이다. 그 뒤에는 SMT 솔버의 바탕을 이루는 알고리즘들을 공부할 시간을 갖겠다.

검증에서는 보통 일차 논리의 부분(fragment)을 사용해 프로그램을 인코딩한다. FOL 은 유구한 역사를 지녔다. FOL 은 매우 일반적인 논리이며, Church (1936)의 증명에 의해 그 만족가능성은 결정 불가능하다. 지난 이십여 년 동안 활발히 연구된 SMT 솔버는 FOL 의 부분들—LRA 및 기타 이론들—을 푸는 것을 목표로 한다. 더 깊이 있는 설명을 원한다면 *Handbook of Satisfiability*를 참고하길 권한다 (Biere et al., 2009).

Chapter 5

Encodings of Neural Networks

이 장의 목표는 신경망을 선형 실수 산술(LRA) 공식으로 변환(번역)하는 것이다. 아이디어는 그 공식이 신경망의 입력–출력 관계를 정확히(또는 건전하게) 포착하도록 만드는 것이다. 이런 공식을 얻고 나면, SMT 솔버를 이용해 올바름 성질을 검증할 수 있다.

5.1 Encoding Nodes

먼저 신경망을 *관계적* 관점에서 특성화한다. 이는 인코딩의 정당성을 확립하는 데 도움이 된다.

Input-output Relations

신경망은 함수 $f_G : \mathbb{R}^n \to \mathbb{R}^m$을 정의하는 그래프 G로 표현된다는 점을 상기하자. f_G의 *입력–출력 관계*를, f_G 실행 후 가능한 모든 입력과 그에 대응하는 출력을 담는 이항 관계 R_G로 정의한다. 형식적으로, f_G의 입력–출력 관계는 다음과 같다:

$$R_G = \{(\boldsymbol{a}, \boldsymbol{b}) \mid \boldsymbol{a} \in \mathbb{R}^n,\ \boldsymbol{b} = f_G(\boldsymbol{a})\}$$

마찬가지로 G의 단일 노드 v의 함수 f_v에 대한 입력–출력 관계를 R_v로 쓴다.

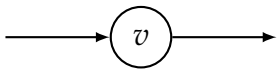
Example 5.A 단순한 함수 $f_G(x) = x + 1$을 생각하자. 이의 입력–출력 관계는

$$R_G = \{(a, a+1) \mid a \in \mathbb{R}\}$$

이다. ■

Encoding a Single Node, Illustrated

단일 노드 v와 연관된 함수 $f_v : \mathbb{R} \to \mathbb{R}$의 경우를 먼저 본다. 입력을 하나만 가지는 노드는 다음처럼 그릴 수 있다 (정의상, 신경망의 노드는 오직 하나의 실수값 출력을 낸다):



$f_v(x) = x + 1$이라고 하자. 그러면 관계 $R_v = \{(a, a+1) \mid a \in \mathbb{R}\}$를 모형화하기 위해 LRA 에서 다음 공식을 만들 수 있다:

$$F_v \triangleq v^{\mathsf{o}} = v^{\mathsf{in},1} + 1$$

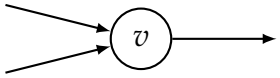
여기서 v^{o}와 $v^{\mathsf{in},1}$은 실수 변수이다. 기호 v^{o}는 노드 v의 출력을, $v^{\mathsf{in},1}$은 그 첫 번째 입력 (입력이 하나뿐이다)을 나타낸다.

F_v의 모델들은 모두 다음 꼴이다:

$$\{v^{\mathsf{in},1} \mapsto a,\ v^{\mathsf{o}} \mapsto a + 1\}$$

임의의 실수 a에 대해 그렇다. R_v의 원소들과 F_v의 모델들 사이에 명확한 일대일 대응이 있음을 볼 수 있다.

이제 입력이 둘인 노드 v를 보자. $f_v(\boldsymbol{x}) = x_1 + 1.5x_2$라고 가정하자.



이때의 인코딩 F_v는 다음과 같다:

$$F_v \triangleq v^{\mathsf{o}} = v^{\mathsf{in},1} + 1.5v^{\mathsf{in},2}$$

입력 벡터의 성분 x_1, x_2가 두 개의 실수 변수 $v^{\mathsf{in},1}$, $v^{\mathsf{in},2}$에 각각 대응함을 관찰하자.

Encoding a Single Node, Formalized

몇 가지 예를 보았으니, 이제 임의의 노드 v의 연산 f_v를 형식적으로 인코딩해 보자. $f_v : \mathbb{R}^{n_v} \to \mathbb{R}$가 다음 꼴의 구간별 선형(piecewise-linear) 함수라고 가정한다:

$$f(\boldsymbol{x}) = \begin{cases} \sum_j c_j^1 \cdot x_j + b^1 & \text{if } S_1 \\ \qquad\vdots & \\ \sum_j c_j^l \cdot x_j + b^l & \text{if } S_l \end{cases}$$

여기서 j는 1부터 n_v까지 범위를 돈다. 또한 각 조건 S_i는 입력 $\boldsymbol{x}$의 성분들 위의 LRA 공식으로 정의된다고 가정한다. 이제 인코딩은 다음과 같다:

$$F_v \triangleq \bigwedge_{i=1}^{l} \left[S_i \Rightarrow \left(v^{\mathsf{o}} = \sum_{j=1}^{n_v} c_j^i \cdot v^{\mathsf{in},j} + b^i \right) \right]$$

이 인코딩은 여러 개의 *if* 문을 결합한 것으로 생각할 수 있다: S_i가 참이면 v^{o}가 i번째 식과 같다는 뜻이다. 함의(⇒)는 조건부를 모형화하게 해 준다. 함의의 왼편이 조건, 오른편이 대입이다. 바깥의 큰 논리곱 $\bigwedge_{i=1}^{l}$은 if 문들을 결합한다: "*만약 S_1이면 … 그리고 S_2이면 … 그리고 S_3이면 …*"

Example 5.B 위 인코딩은 지수와 첨자가 많아 너무 일반적이다. 간단하고 실용적인 예, ReLU를 보자:

$$\text{relu}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leqslant 0 \end{cases}$$

f_v가 ReLU인 노드 v는 다음처럼 인코딩된다:

$$F_v \triangleq (\underbrace{v^{\mathsf{in},1} > 0}_{x>0} \Rightarrow v^{\mathsf{o}} = v^{\mathsf{in},1}) \wedge (\underbrace{v^{\mathsf{in},1} \leqslant 0}_{x\leqslant 0} \Rightarrow v^{\mathsf{o}} = 0)$$

■

Soundness and Completeness

위 인코딩은 구간별 선형 노드의 의미론을 정확히 포착한다. 이를 형식화하자: 구간별 선형 함수 f_v를 갖는 어떤 노드 v를 고정하자. 위에서 정의한 대로 그 인코딩을 F_v라 하자.

첫째, 인코딩은 *건전*하다(sound): 노드의 모든 실행이 공식 F_v의 어떤 모델로 잡힌다. 비형식적으로, 건전성은 인코딩이 f_v의 어떤 동작도 놓치지 않는다는 뜻이다. 형식적으로, $(\boldsymbol{a}, b) \in R_v$라 하고

$$I = \{v^{\mathsf{in},1} \mapsto a_1,\ \ldots, v^{\mathsf{in},n} \mapsto a_n,\ v^{\mathsf{o}} \mapsto b\}$$

라 하자. 그러면 $I \models F_v$이다.

둘째, 인코딩은 *완전*하다(complete): F_v의 모든 모델이 f_v의 어떤 동작에 대응한다. 비형식적으로, 완전성은 인코딩이 촘촘하여 f_v에 없는 새 동작을 도입하지 않는다는 뜻이다. 형식적으로, F_v의 어떤 모델이

$$I = \{v^{\mathsf{in},1} \mapsto a_1,\ \ldots, v^{\mathsf{in},n} \mapsto a_n,\ v^{\mathsf{o}} \mapsto b\}$$

라 하자. 그러면 $(\boldsymbol{a}, b) \in R_v$이다.

5.2 **Encoding a Neural Network**

단일 노드를 인코딩하는 법을 보았다. 이제 전체 그래프를 인코딩할 준비가 되었다. 인코딩은 두 부분으로 나뉜다: (1) 모든 노드의 의미론을 인코딩하는 공식, (2) 노드들을 연결하는, 즉 간선들을 인코딩하는 공식.

Encoding the Nodes

신경망은 $G = (V, E)$인 그래프이며, V에는 아무 연산도 하지 않는 입력 노드 V^{in}가 포함된다는 점을 상기하자. G의 비입력 노드에 대한 인코딩을 결합하면 다음 공식을 얻는다:

$$F_V \triangleq \bigwedge_{v \in V \setminus V^{\mathsf{in}}} F_v$$

다시 말해, 큰 논리곱은 "*노드 v_1의 출력은 … 그리고 노드 v_2의 출력은 … 그리고 …*"을 뜻한다. 하지만 이 공식만으로는 무의미하다: 각 노드의 입력–출력 관계만 인코딩할 뿐, 이들 사이의 연결은 인코딩하지 않기 때문이다!

Encoding the Edges

이제 간선을 인코딩하자. 각 노드에 대해, 들어오는 모든 간선을 인코딩하는 방식으로 진행한다. 임의의 노드 $v \in V \setminus V^{\mathsf{in}}$를 고정하자. 목적지가 v인 모든 간선의 순서열을 $(v_1, v), \ldots (v_n, v)$라 하자. Section 2.5에서 간선에 전순서가 있다고 가정했음을 기억하자. 이 순서를 두는 이유는 어떤 들어오는 간선이 노드의 어느 입력에 꽂히는지 알기 위함이다. 비형식적으로, 간선 관계 E는 노드 v에 꽂을 여러 가닥의 선(wire)을 제공하고; 그 순서는 어디에 꽂을지를 말해 준다—첫 번째 선은 첫 번째 소켓, 두 번째 선은 두 번째 소켓, 등등.

이제 v의 간선에 대한 공식을 다음처럼 정의할 수 있다:

$$F_{\circ \to v} \triangleq \bigwedge_{i=1}^{n} v^{\mathsf{in},i} = v_i^{\mathsf{o}}$$

직관적으로, 각 간선 (v_i, v)에 대해 노드 v_i의 출력을 v의 i번째 입력에 연결한다. 이제 모든 비입력 노드에 대한 *들어오는* 간선들을 논리곱으로 묶어 F_E을 정의한다:

$$F_E \triangleq \bigwedge_{v \in V \setminus V^{\mathsf{in}}} F_{\circ \to v}$$

Putting it all Together

노드와 간선을 인코딩하는 법을 보았으니, 더 이상 인코딩할 것은 없다! 이제 합치자. 그래프 $G = (V, E)$가 주어졌을 때, 그 인코딩을 다음과 같이 정의한다:

$$F_G \triangleq F_V \wedge F_E$$

단일 노드 인코딩에서와 마찬가지로, 건전성과 완전성을 얻는다. G의 입력–출력 관계를 R_G라 하자. 건전성은 F_G가 R_G의 어떤 동작도 놓치지 않음을 뜻한다. 완전성은 F_G의 모든 모델이 G의 입력–출력 동작에 대응함을 뜻한다.

인코딩의 크기는 신경망의 크기(노드 수와 간선 수)에 선형적임에 주의하라. 간단히 말해, 각 노드마다 공식 하나, 각 간선마다 공식 하나를 갖는다. 노드 v의 공식 크기는 (구간별) 선형 함수 f_v의 크기에 선형적이다.

Correctness of the Encoding

다음과 같이 정렬된 입력 노드들을 V^{in}에 갖고 있다고 하자:

$$v_1, \ldots, v_n$$

그리고 다음과 같은 출력 노드들을 V^{o}에 갖고 있다고 하자:

$$v_{n+1}, \ldots, v_{n+m}$$

우리 인코딩은 건전하고 완전하다. 먼저 건전성을 진술하자: $(\boldsymbol{a}, \boldsymbol{b}) \in R_G$라 하고

$$I = \{v_1^{\mathrm{o}} \mapsto a_1, \ldots, v_n^{\mathrm{o}} \mapsto a_n\} \cup \{v_{n+1}^{\mathrm{o}} \mapsto b_1, \ldots, v_{n+m}^{\mathrm{o}} \mapsto b_m\}$$

라 하자. 그러면 어떤 I'가 존재하여 $I \cup I' \models F_G$가 된다.

단일 노드와 달리, F_G의 모델에는 네트워크의 입력과 출력뿐 아니라 중간 노드들에 대한 대입도 포함된다는 점에 주목하라. 입력/출력 노드의 출력값에 대입을 주는 I와, 모든 노드의 입력과 출력에 값을 주되 I와 영역이 겹치지 않는 I'가 이를 처리한다.

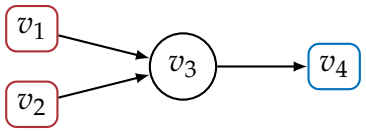
마찬가지로 완전성은 다음과 같이 진술된다: F_G의 어떤 모델이

$$I = \{v_1^{\mathrm{o}} \mapsto a_1, \ldots, v_n^{\mathrm{o}} \mapsto a_n\} \cup \{v_{n+1}^{\mathrm{o}} \mapsto b_1, \ldots, v_{n+m}^{\mathrm{o}} \mapsto b_m\} \cup I'$$

라 하자. 그러면 $(\boldsymbol{a}, \boldsymbol{b}) \in R_G$이다.

An Example Network and its Encoding

추상 수학은 이쯤에서 충분하다. 구체적 예시 신경망 G를 보자.



$f_{v_3}(\boldsymbol{x}) = 2x_1 + x_2$이고 $f_{v_4}(x) = \mathrm{relu}(x)$라고 가정하자.

비입력 노드들에 대한 공식을 먼저 만든다:

$$\begin{aligned} F_{v_3} &\triangleq v_3^{\mathrm{o}} = 2v_3^{\mathsf{in},1} + v_3^{\mathsf{in},2} \\ F_{v_4} &\triangleq (v_4^{\mathsf{in},1} > 0 \implies v_4^{\mathrm{o}} = v_4^{\mathsf{in},1}) \ \wedge \ (v_4^{\mathsf{in},1} \leqslant 0 \implies v_4^{\mathrm{o}} = 0) \end{aligned}$$

다음으로 간선 공식을 만든다:

$$\begin{aligned} F_{\circ \to v_3} &\triangleq (v_3^{\mathsf{in},1} = v_1^{\mathrm{o}}) \wedge (v_3^{\mathsf{in},2} = v_2^{\mathrm{o}}) \\ F_{\circ \to v_4} &\triangleq v_4^{\mathsf{in},1} = v_3^{\mathrm{o}} \end{aligned}$$

마지막으로 위 공식들을 모두 논리곱해 G의 전체 인코딩을 얻는다:

$$F_G \triangleq \underbrace{F_{v_3} \wedge F_{v_4}}_{F_V} \wedge \underbrace{F_{\circ \to v_3} \wedge F_{\circ \to v_4}}_{F_E}$$

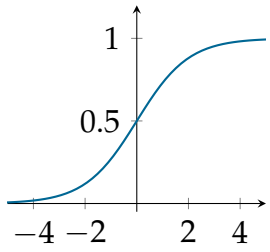


Figure 5.1 Sigmoid function

5.3 Handling Non-linear Activations

위에서는 모든 노드가 구간별 선형 함수에 연관된다고 가정했기 때문에, 선형 실수 산술에서 그 의미론을 정확히 포착할 수 있었다. 그렇다면 sigmoid나 tanh 같은 비구간별-선형 활성화는 어떻게 다룰까? 한 가지 방법은 그 동작을 *과대근사(overapproximation)*로 인코딩하는 것이다. 이 경우 건전성은 갖지만 *완전성*은 갖지 않는다. 앞에서 보았듯, 건전성은 인코딩으로 올바름 성질의 *증명*을 찾을 수 있음을, 완전성은 올바름 성질의 *반례*를 찾을 수 있음을 의미한다. 따라서 활성화 함수를 과대근사하면, 반례를 포기하는 셈이다.

Handling Sigmoid

구체적 예로 sigmoid 활성화를 보자:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

이는 Figure 5.1에 나와 있다. sigmoid 함수는 (엄밀히) 단조 증가하므로, $a_1 < a_2$이면 $\sigma(a_1) < \sigma(a_2)$임을 안다. 이에 따라, 다음처럼 σ의 동작을 과대근사할 수 있다: 입력이 a_1과 a_2 사이이면, 함수의 출력은 $\sigma(a_1)$과 $\sigma(a_2)$ 사이의 *임의의* 값이 될 수 있다고 말하는 것이다.

Figure 5.2을 보자. 여기서는 x 좌표가 $-1, 0, 1$인 세 점을 빨간색으로 골랐다. 빨간 사각형들은, x의 두 값 사이에서 sigmoid 함수 출력의 하한과 상한을 정의한다. 예컨대 입력이 0과 1 사이이면, 함수의 출력은 0.5와 0.73 사이의 어떤 값도 될 수 있다. 입력이 1보다 크면, 출력은 0.73과 1 사이임을 안다(σ의 치역은 위로 1로 유계).

어떤 노드 v에 대해 f_v가 sigmoid 활성화라고 하자. 그러면 Figure 5.2의 근사에 따른 한 가지 인코딩은 다음과 같다:

$$\begin{aligned} F_v \triangleq\ & (v^{\mathsf{in},1} \leqslant -1 \implies 0 < v^{\mathsf{o}} \leqslant 0.26) \\ & \wedge (-1 < v^{\mathsf{in},1} \leqslant 0 \implies 0.26 < v^{\mathsf{o}} \leqslant 0.5) \\ & \wedge (0 < v^{\mathsf{in},1} \leqslant 1 \implies 0.5 < v^{\mathsf{o}} \leqslant 0.73) \\ & \wedge (v^{\mathsf{in},1} > 1 \implies 0.73 < v^{\mathsf{o}} < 1) \end{aligned}$$

각 항은 입력 구간(함의의 왼편)과 그 구간에서 가능한 출력(오른편)을 지정한다. 예컨대 첫 항은 입력이 $\leqslant -1$이면 출력이 0과 0.26 사이의 어떤 값도 될 수 있음을 지정한다.

Handling any Monotonic Function

위 과정을 임의의 단조(증가 또는 감소) 함수 f_v로 일반화할 수 있다.

f_v가 단조 증가한다고 가정하자. 실수 $c_1 < \cdots < c_n$의 수열을 택할 수 있다. 그러면 다음 인코딩을 만들 수 있다:

$$\begin{aligned} F_v \triangleq\ & (v^{\mathsf{in},1} \leqslant c_1 \implies lb < v^{\mathsf{o}} \leqslant f_v(c_1)) \\ & \wedge (c_1 < v^{\mathsf{in},1} \leqslant c_2 \implies f_v(c_1) < v^{\mathsf{o}} \leqslant f_v(c_2)) \\ & \vdots \\ & \wedge (c_n < v^{\mathsf{in},1} \implies f_v(c_n) < v^{\mathsf{o}} \leqslant ub) \end{aligned}$$

여기서 lb와 ub는 f_v의 치역의 하한과 상한이다; 예컨대 sigmoid의 경우 각각 0과 1이다. 함수가 무계이면, 제약 $lb \leqslant v^{\mathsf{o}}$와 $v^{\mathsf{o}} \leqslant ub$는 생략할 수 있다.

점을 c_i로 더 많이, 더 촘촘히 고를수록 근사는 좋아진다. 이 인코딩은 건전하지만 *불완전*하다. 활성화 함수가 취할 수 있는 것보다 더 많은 동작을 담기 때문이다. sigmoid 예에서, 입력이 $\geqslant$ 1이면 인코딩은 출력이 0.73과 1 사이의 *임의의* 값이라고 말한다 (Figure 5.2의 가장 오른쪽 음영 영역).

5.4 Encoding Correctness Properties

이제 신경망의 의미론을 논리 제약으로 인코딩하는 법을 보았으니, 본론으로 들어간다: 전체 올바름 성질을 인코딩하는 것이다.

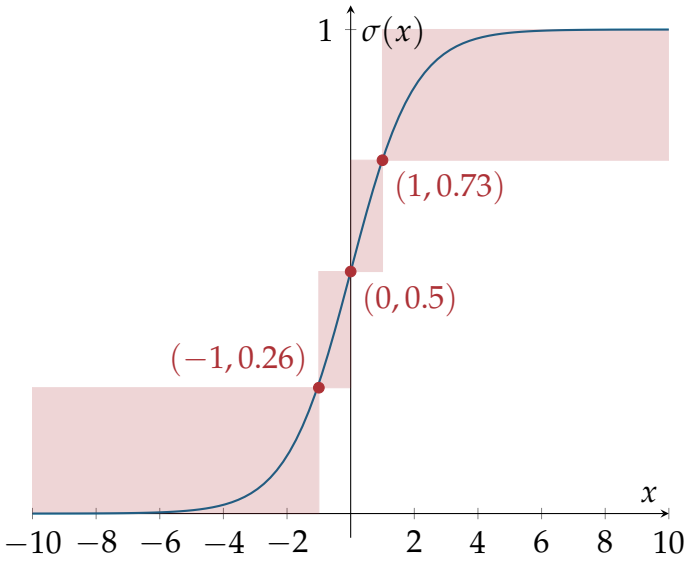


Figure 5.2 Sigmoid function with overapproximation

Checking Robustness Example

일반형을 보기 전에 구체적 예로 시작하자. 이진 분류기 $f_G : \mathbb{R}^n \to \mathbb{R}^2$를 정의하는 신경망 G가 있다고 하자. 신경망 f_G는 0과 1 사이의 실수 벡터(각 픽셀의 세기: 검정에서 흰색)를 입력으로 받아 그레이스케일 이미지를 표현하고, 이미지가 *고양이*인지 *개*인지 예측한다. *고양이*로 올바르게 분류된 이미지 $\boldsymbol{c}$가 있다고 하자. $\boldsymbol{c}$의 밝기를 조금 섭동해도 예측이 변하지 않음을 증명하고자 한다. 이를 다음과 같이 형식화한다:

$$\begin{gathered}\{\ |\boldsymbol{x} - \boldsymbol{c}| \leqslant \mathbf{0.1}\ \} \\ \boldsymbol{r} \leftarrow f_G(\boldsymbol{x}) \\ \{\ r_1 > r_2\ \}\end{gathered}$$

여기서 첫 번째 출력 r_1은 *고양이*의 확률, r_2는 *개*의 확률이다.

이 올바름 성질 인코딩의 높은 수준 직관은 성질을 적은 그대로 따른다. 이 진술을 검사하기 위해 만들어 내는 공식을 *검증 조건*(verification condition, VC)이라 하며, 대략 다음과 같은 꼴이다:

$$(\text{precondition} \wedge \text{neural network}) \Longrightarrow \text{postcondition}$$

이 공식이 *타당*하면, 올바름 성질이 성립한다.

예제를 위해, 신경망의 입력 노드가 $\{v_1, \ldots, v_n\}$, 출력 노드가 $\{v_{n+1}, v_{n+2}\}$라고 하자. 또한 앞에서 본 대로 F_G가 네트워크를 인코딩한다고 하자. 올바름 성질을 다음처럼 인코딩한다:

$$\underbrace{\left(\bigwedge_{i=1}^{n} |x_i - c_i| \leqslant 0.1\right)}_{\text{precondition}} \wedge \underbrace{F_G}_{\text{network}} \wedge \underbrace{\left(\bigwedge_{i=1}^{n} x_i = v_i^{\circ}\right)}_{\text{network input}} \wedge \underbrace{\left(r_1 = v_{n+1}^{\circ} \wedge r_2 = v_{n+2}^{\circ}\right)}_{\text{network output}}$$

$$\implies \underbrace{r_1 > r_2}_{\text{postcondition}}$$

구성은 다음과 같다:

- 전제는 LRA 공식으로 직접 번역한다. LRA 공식은 벡터 연산을 기본적으로 지원하지 않으므로, 벡터를 성분 스칼라들로 분해한다. 절댓값 $|\cdot|$ 연산은 LRA 에 기본 제공되지 않지만 걱정할 필요 없다. 실제로 다음처럼 인코딩된다: $|x| \leqslant 5$ 같은 선형 부등식은 $x \leqslant 5 \wedge -x \leqslant 5$로 쓸 수 있다.
- 네트워크는 앞서 Section 5.2에서 본 대로 F_G라는 공식으로 인코딩한다. 이때 요령은, F_G의 변수들을 입력 $\boldsymbol{x}$와 출력 $\boldsymbol{r}$에 연결해야 한다는 점이다. 이는 "network input"과 "network output"으로 표시한 두 부분 공식으로 표현한다.
- 사후조건은 있는 그대로 인코딩한다.

Encoding Correctness, Formalized

올바름 성질은 다음 꼴이다:

$$\begin{gathered}\{\, P \,\} \\ \boldsymbol{r_1} \leftarrow f_{G_1}(\boldsymbol{x_1}) \\ \boldsymbol{r_2} \leftarrow f_{G_2}(\boldsymbol{x_2}) \\ \vdots \\ \boldsymbol{r_l} \leftarrow f_{G_l}(\boldsymbol{x_l}) \\ \{\, Q \,\}\end{gathered}$$

책의 예시 대부분은 단일 신경망 f_G를 다루지만, Chapter 3에서 보았듯 우리의 성질은 여러 네트워크의 조합도 허용한다.

전제와 사후조건이 LRA 로 인코딩 가능하다고 가정하자. 그러면 검증 조건을 다음처럼 인코딩한다:

$$\left(P \wedge \bigwedge_{i=1}^{l} F_i\right) \Longrightarrow Q$$

여기서 F_i는 i번째 대입 $\boldsymbol{r_i} \leftarrow f_{G_i}(\boldsymbol{x_i})$의 인코딩이다. 대입 인코딩 F_i는 신경망 인코딩 F_{G_i}에, 각각 입력과 출력인 $\boldsymbol{x_i}, \boldsymbol{r_i}$와의 *연결*을 결합한다:

$$F_i \triangleq F_{G_i} \wedge \left(\bigwedge_{j=1}^{n} x_{i,j} = v_i^{\mathrm{o}}\right) \wedge \left(\bigwedge_{j=1}^{m} r_{i,j} = v_{n+j}^{\mathrm{o}}\right)$$

여기서 두 가지를 가정한다:

- G_i 인코딩의 입력/출력 변수는 각각 $v_1, \ldots, v_n$ 및 $v_{n+1}, \ldots, v_{n+m}$이다.
- 각 그래프 G_i는 고유한 노드들(따라서 고유한 입력–출력 변수들)을 갖는다.

비형식적으로, 우리의 인코딩 $\left(P \wedge \bigwedge_{i=1}^{l} F_i\right) \Longrightarrow Q$는 다음을 말한다: "*전제가 참이고 그리고 l개의 네트워크를 모두 실행하면, 사후조건이 참이어야 한다*"

Soundness and Completeness

올바름 성질 하나를 공식 F로 인코딩했다고 하자. 그러면 다음과 같은 건전성 보증을 갖는다: F가 타당하면, 올바름 성질은 참이다.

완전성은 모든 함수가 LRA 로 인코딩 가능한지에 달려 있다. 모든 함수가 LRA 로 인코딩 가능하다고 가정하면, F가 타당하지 않을 때 $\neg F$의 모델 I가 있음을 안다. 이 모델은 올바름 성질의 반례이다. 이 모델에서, 사후조건을 만족하지 않는 출력을 초래하는 입력 변수들의 값을 읽어낼 수 있다. 예를 통해 가장 잘 보인다:

Example 5.C 다음의 단순한 올바름 성질을 보자. 여기서 $f(x) = x$이다:

$$\{\ |x - 1| \leqslant 0.1\ \}$$
$$r \leftarrow f(\boldsymbol{x})$$
$$\{\ r \geqslant 1\ \}$$

이 성질은 참이 아니다. $x = 0.99$로 두자. 전제를 만족한다. 하지만 $f(0.99) = 0.99$로, 1보다 작다. 이 성질에 대한 공식 F를 인코딩하면, $\neg F$의 어떤 모델 I가 존재하고 그 모델에서 x는 0.99로 배정된다. ■

Looking Ahead

아, 꽤 벅찬 장이었다! 끝까지 따라와 줘서 고맙다. 우리는 신경망을, 그 모든 화려함과 함께, 공식들로 번역했다. 이어지는 장들에서는 이 공식들의 만족가능성을 확인하는 알고리즘들을 공부할 것이다.

내가 아는 한, 검증을 위한 제약으로 신경망을 인코딩한 최초의 논문은 흥미 폭발 이전에 나온 Pulina and Tacchella (2010)이다. 강건성 검증을 제약 풀이 문제로 본 최초의 연구는 Bastani et al. (2016)이다.

우리의 sigmoid 인코딩은 Ehlers (2017)를 따른다. MILP 인코딩을 다룬 여러 논문들이 있으며, 우리가 제시한 것과 유사하다 (Tjeng et al., 2019a). MILP 에서는 논리합이 없으므로, $\{0,1\}$ 값을 취하는 정수로 논리합을 모사한다. LRA 와 MILP 인코딩의 주요 이슈는 논리합이다; 논리합이 없으면 문제는 다항시간에 풀린다. 논리합은 주로 ReLU 때문에 생긴다. ReLU의 출력이 > 0이면 *활성(active)*, 아니면 *비활성*이라 하자. 전제가 허용하는 모든 가능한 입력에 대해 ReLU가 항상 활성/비활성이라면, 논리합을 제거할 수 있고, 그것을 $f(x) = 0$(비활성) 또는 $f(x) = x$(활성)으로 다룰 수 있다. 이 아이디어를 염두에 두면, 검증을 단순화하는 두 가지 요령이 있다: (1) 어떤 ReLU가 활성/비활성인지 가볍게 판정하는 기법을 고안한다. 책 3부의 추상화 기반 검증 기법을 사용할 수 있다. (2) 보통 신경망을 학습할 때는 데이터 정확도를 최대화하려고 한다; 여기에 더해, 대부분의 ReLU가 항상 활성/비활성인 신경망이 되도록 학습을 편향시킬 수 있다 (Tjeng et al., 2019b).

실제로 신경망은 유한 정밀도 산술로 구현되며, 실수는 부동소수점, 고정소수점, 혹은 심지어 기계 정수로 근사된다. 일부 논문은 검증 결과가 네트워크의 부동소수점 구현에 대해서도 성립함을 주의 깊게 보장한다 (Katz et al., 2017). 또한, LRA 에서 검증된 신경망이 비트 수준 동작을 고려하면 실제로는 강건하지 않을 수도 있음을 보인 최근 논문도 있다 (Jia and Rinard, 2020b). 또한 여러 논문들이 LRA 대신 명제 논리를 사용한 비트 수준 신경망 검증을 다루었다 (Jia and Rinard, 2020a; Narodytska et al., 2018).

Chapter 6

DPLL Modulo Theories

앞 장에서는 검증 문제를 논리 공식의 만족가능성(satisfiability) 검사 문제로 환원하는 방법을 보았다. 그렇다면 실제로는 만족가능성을 어떻게 검사할까? 이 장에서는 불(Boolean) 공식의 만족가능성 검사를 위해 수십 년 전에 개발된 DPLL (Davis–Putnam–Logemann–Loveland) 알고리즘을 살펴본다. 그리고 이 DPLL 을, 이론 위의 1차 논리 공식을 다룰 수 있도록 확장한 방법을 본다.

이 알고리즘들은 현대의 SAT 및 SMT 솔버의 기반을 이룬다. 이 장에서는 DPLL 을 *완전한* 수준으로 설명하여, 독자가 장을 따라가며 동작하는 알고리즘을 구현할 수 있도록 할 것이다. 다만 실제로 DPLL 이 *매우* 잘 동작하게 해 주는 수많은 자료구조, 구현 요령, 휴리스틱들은 여기서 다루지 않는다(장 끝에서 추가 자료를 안내한다).

6.1 Conjunctive Normal Form (CNF)

DPLL 의 목표는 불 공식 F를 받아 그것이 SAT 인지 UNSAT 인지 결정하는 것이다. 만약 F가 SAT 이라면, DPLL 은 F의 모델도 함께 반환해야 한다. 우선 DPLL 이 입력으로 기대하는 공식의 *모양*부터 이야기하자.

DPLL 은 공식을 *합정규형*(CNF)으로 받는다. 다행히 모든 불 공식은 CNF 와 동치인 공식으로 다시 쓸 수 있다 (어떻게 하는지는 이 장의 뒤에서 본다). CNF 인 공식 F는 다음 꼴이다:

$$C_1 \wedge \cdots \wedge C_n$$

여기서 각 부분 공식 C_i를 *절(clause)*이라고 부르며, 다음 꼴이다:

$$\ell_1 \vee \cdots \vee \ell_{m_i}$$

각 ℓ_i는 *리터럴(literal)*이라 부르며, 변수(예: p) 또는 그 부정(예: $\neg p$) 중 하나다.

Example 6.A 다음은 각기 두 개의 리터럴을 포함하는 두 절로 이루어진 CNF 공식이다:

$$(p \vee \neg r) \wedge (\neg p \vee q)$$

다음 공식은 CNF *가 아니다*:

$$(p \wedge q) \vee (\neg r)$$

■

6.2 The DPLL Algorithm

만족가능성을 결정하는 *완전히* 단순한 방법은 가능한 모든 해석을 전부 시도해 보며 모델인지 확인하는 것이다. 물론 변수 수에 대해 지수적으로 많은 해석이 있다. DPLL은 이렇게 완전히 맹목적인 탐색을 피하려 하지만, 운이 나쁘면 결국 가능한 모든 해석을 지수적으로 나열하는 데 이른다—결국 만족가능성은 전형적인 NP-완전 문제이기 때문이다.

DPLL은 두 단계를 번갈아 수행한다: *추론(deduction)*과 *탐색(search)*. 추론은 논리 법칙들을 이용해 공식을 단순화하려 한다. 탐색은 그저 어떤 해석을 찾기 위해 탐색한다.

Deduction

DPLL의 추론 부분은 *불 상수 전파*(BCP, Boolean constant propagation)라고 한다. 다음과 같은 CNF 공식을 생각하자:

$$(\ell) \wedge C_2 \wedge \cdots C_n$$

첫 번째 절이 하나의 리터럴만으로 되어 있음을 보라—이를 *단위 절(unit clause)*이라 부른다. 이 공식의 모든 모델은 분명히 ℓ에 true를 할당해야 한다: 구체적으로, ℓ이 변수 p라면 p는 true가 되어야 하고, ℓ이 부정 $\neg p$라면 p는 false가 되어야 한다.

DPLL 의 BCP 단계는 모든 단위 절을 찾아 그 리터럴을 true로 바꾸어 버린다. BCP 는 공식이 CNF 인 덕분에 가능하며, SAT 이나 UNSAT 임을 증명하는 데 상당히 효과적일 수 있다.

Example 6.B 다음 공식을 보자

$$F \triangleq (p) \wedge (\neg p \vee r) \wedge (\neg r \vee q)$$

BCP 는 먼저 단위 절 (p)를 찾아 p에 true를 할당한다. 그러면 공식은 다음처럼 된다:

$$\begin{aligned} & (\mathsf{true}) \wedge (\neg \mathsf{true} \vee r) \wedge (\neg r \vee q) \\ \equiv\ & (r) \wedge (\neg r \vee q) \end{aligned}$$

분명 BCP 의 일이 끝난 것은 아니다: 단순화 결과 또 다른 단위 절 (r)이 생겼다. BCP 는 r을 true로 두어 다음 공식을 얻는다:

$$\begin{aligned} & (\mathsf{true}) \wedge (\neg \mathsf{true} \vee q) \\ \equiv\ & (q) \end{aligned}$$

마지막으로 단위 절 (q) 하나만 남는다. 따라서 BCP 는 q를 true로 두고, 최종 공식 true 를 얻는다. 이는 F가 SAT 임을 의미하며, $\{p \mapsto \mathsf{true}, q \mapsto \mathsf{true}, r \mapsto \mathsf{true}\}$ 가 모델이다. ■

위 예에서 BCP 는 F의 만족가능성을 보였다. BCP 는 공식을 false로 단순화하여 UNSAT 임도 증명할 수 있다. 하지만 단위 절을 찾지 못하면 BCP 는 막힐 수 있다. 이때 DPLL 은 탐색을 수행한다.

Deduction + Search

Algorithm 1 에 DPLL 알고리즘 전체를 보였다. 알고리즘은 CNF 인 공식 F를 입력으로 받는다.

첫 부분은 BCP 를 수행한다: 단위 절이 더는 없을 때까지 공식을 계속 단순화한다. 기호 $F[\ell \mapsto \mathsf{true}]$는 F에서 ℓ의 모든 출현을 true로 바꾸고 결과 공식을 단순화한다는 뜻이다. 구체적으로, ℓ이 변수 p이면 p의 모든 출현을 true로, ℓ이 부정 $\neg p$이면 p의 모든 출현을 false로 바꾼다.

Algorithm 1: DPLL

Data: CNF 꼴의 공식 F
Result: $I \models F$ 또는 UNSAT

▷ 불 상수 전파 (BCP)
while *F 안에 단위 절 (ℓ)이 있을 때* **do**
  F를 $F[\ell \mapsto \mathsf{true}]$로 둔다

if *F가* true **then return** SAT

▷ 탐색
for *F의 모든 변수 p에 대해* **do**
  If DPLL$(F[p \mapsto \mathsf{true}])$가 SAT **then return** SAT
  If DPLL$(F[p \mapsto \mathsf{false}])$가 SAT **then return** SAT
return UNSAT

▷ 결과가 SAT 일 때 DPLL 이 반환하는 모델 I는, BCP 와 탐색이 SAT 을 반환하기까지 수행한 변수 대입들($[l \mapsto \cdot]$, $[p \mapsto \cdot]$ 꼴)의 열에 암묵적으로 저장된다

BCP 가 끝나면 알고리즘은 공식이 true인지 확인한다. 그렇다면 BCP 가 공식을 SAT 임을 증명한 것이다.

BCP 만으로 SAT 을 증명하지 못하면, DPLL 은 탐색 단계로 간다: 변수들을 하나씩 골라 그들을 true 또는 false로 바꾸어 보며, 그 결과 공식을 인자로 DPLL 을 재귀적으로 호출한다. 탐색에서 변수를 고르는 순서는 DPLL 의 성능에 매우 중요하다. 변수 선택에 관한 연구가 많다. 유명한 휴리스틱 중 하나는 지속적으로 갱신되는 점수표를 유지하며, 점수가 높은 변수를 먼저 고르는 방식이다.

Algorithm 1 은 제시된 그대로라면 입력 공식이 만족가능할 때 SAT 을 반환하지만 모델은 직접 반환하지 않는다. 결과가 SAT 일 때 DPLL 이 반환하는 모델 I는, BCP 와 탐색이 SAT 을 반환하기까지 수행한 변수 대입들($[l \mapsto \cdot]$ 및 $[p \mapsto \cdot]$ 꼴)의 열에 암묵적으로 담겨 있다. 알고리즘은 가능한 모든 만족 대입을 소진하면 UNSAT 을 반환한다.

Example 6.C DPLL 에게 다음 공식을 주자

$$F \triangleq (p \vee r) \wedge (\neg p \vee q) \wedge (\neg q \vee \neg r)$$

첫 번째 재귀 수준 DPLL 은 BCP 를 시도하지만 단위 절을 찾지 못한다. 이어서 탐색을 수행한다. 탐색이 변수 p를 골라

$$F_1 = F[p \mapsto \mathsf{true}] = q \wedge (\neg q \vee \neg r)$$

에 대해 DPLL 을 재귀 호출하여 p를 true로 두었다고 하자.

두 번째 재귀 수준 이제 DPLL 은 F_1에 BCP 를 시도한다. 먼저 q를 true로 두어

$$F_2 = F_1[q \mapsto \mathit{true}] = (\neg r)$$

를 얻는다. 이어서 r을 false로 두어

$$F_3 = F_2[r \mapsto \mathsf{false}] = \mathsf{true}$$

를 얻는다. F_3가 true이므로, DPLL 은 SAT 을 반환한다. 암묵적으로, DPLL 은 F의 모델을 축적했다:

$$\{p \mapsto \mathsf{true}, q \mapsto \mathsf{true}, r \mapsto \mathsf{false}\}$$

■

Partial Models

DPLL 이 SAT 으로 종료하더라도 공식의 모든 변수에 값을 할당하지 않을 수 있다. 이 때의 모델을 *부분 모델(partial model)*이라 부른다. 부분 모델을 임의의 방식으로 나머지 변수들에 값을 더 할당해 *확장*하더라도, 여전히 공식의 모델이 된다.

Example 6.D 다음의 간단한 공식을 보자:

$$F \triangleq p \wedge (q \vee p \vee \neg r) \wedge (p \vee \neg q)$$

DPLL 은 먼저 BCP 를 적용하여 단위 절 p를 true로 둔다. 그러고 나면 나머지 공식은 true로 단순화된다. 이는 q와 r이 쓸모없는 변수라는 뜻—그들에게 어떤 해석을 주더라도, p가 true로만 할당되어 있으면 모델이 된다. 따라서 $I = \{p \rightarrow \mathsf{true}\}$를 F의 부분 모델이라 부른다. 형식적으로, $\mathsf{eval}(I(F)) = \mathsf{true}$이다. ■

6.3 DPLL Modulo Theories

지금까지 불 공식의 만족가능성을 DPLL 이 어떻게 결정하는지 보았다. 이제 *이론*을 *모듈로 한* DPLL, 즉 DPLL^T 을 소개한다. 이는 DPLL 을 확장하여, 예컨대 LRA 같은 산술 이론 위의 공식을 다룰 수 있게 한다. DPLL^T 의 핵심 아이디어는, 공식을 완전히 불 공식인 양 다루는 것에서 시작해 점진적으로 더 많은 *이론* 정보를 더해 가며 공식이 SAT 인지 UNSAT 인지 결론낼 수 있을 때까지 정제한다는 점이다. 우선 공식의 불 추상화 (Boolean abstraction)를 정의한다.

Boolean Abstraction

설명을 위해 앞 장들과 마찬가지로 LRA 의 공식을 다룬다고 하자. 다음 LRA 공식을 생각하자:

$$F \triangleq (x \leqslant 0 \vee x \leqslant 10) \wedge (\neg x \leqslant 0)$$

F의 불 추상화(부울 추상화) F_B는, 공식 안의 모든 서로 다른 선형 부등식을 특수한 불 변수로 치환하여 얻는 공식이다:

$$F^B \triangleq (p \vee q) \wedge (\neg p)$$

부등식 $x \leqslant 0$은 p로 *추상화* 되고, $x \leqslant 10$은 q로 추상화된다. $x \leqslant 0$의 두 출현이 같은 불 변수로 바뀌었음을 주목하라(우리 설명의 정당성을 위해 꼭 그래야 하는 것은 아니다). 또한 위첨자 T를 써서 불 공식을 다시 이론 공식으로 사상하기도 한다. 예컨대 $(F^B)^T$는 F다.

이 과정을 *추상화* 라고 부르는 이유는 그 과정에서 제약이 손실되기 때문이다; 특히 서로 다른 부등식 사이의 관계가 사라진다. 형식적으로, 만약 F^B가 UNSAT 이면, F도 UNSAT 이다. 그러나 그 역은 성립하지 않는다: F^B가 SAT 이라고 해서 F가 SAT 이라는 뜻은 아니다.

Example 6.E $F \triangleq x \leqslant 0 \wedge x \geqslant 10$을 생각하자. 이 공식은 명백히 UNSAT 이다. 하지만 그 추상화 $p \wedge q$는 SAT 이다. ■

Lazy DPLL Modulo Theories

DPLL^T 알고리즘은 LRA 같은 어떤 이론 위의 공식 F를 받아 그 만족가능성을 결정한다. DPLL^T 은 *이론 솔버*에 접근할 수 있다고 가정한다. 이론 솔버는 (예컨대) 선형 부등식

Algorithm 2: DPLL^T

Data: 이론 T 위의 CNF 꼴 공식 F
Result: $I \models F$ 또는 UNSAT

F의 추상화를 F^B라 두자
while true **do**
  If DPLL(F^B)가 UNSAT **then return** UNSAT
  DPLL(F^B)가 반환한 모델을 I라 두자
  I가 공식으로 표현되어 있다고 가정하자
  if I^T*가 (이론 솔버로) 만족가능하다* **then**
    return SAT 및 이론 솔버가 반환한 모델
  else
    F^B를 $F^B \wedge \neg I$로 둔다

들의 논리곱을 받아 그것이 만족가능한지 검사한다. 어떤 의미에서, 이론 솔버는 논리곱을, DPLL 은 논리합을 맡는다. LRA 의 경우, 이론 솔버는 다음 장에서 볼 심플렉스 (Simplex) 알고리즘이 될 수 있다.

Algorithm 2 에 보인 DPLL^T 의 동작은 다음과 같다: 먼저, 순수 DPLL 을 이용해 추상화 F^B가 UNSAT 인지 확인한다. 위에서 논한 추상화의 성질에 따라, 그렇다면 F도 UNSAT 라고 선언할 수 있다. 까다로운 부분은 F^B가 SAT 인 경우인데, 이는 반드시 F가 SAT 임을 의미하지 않기 때문이다. 여기서 이론 솔버가 등장한다. DPLL(F^B)가 반환한 모델 I를 이론의 공식 I^T로 사상한다; 예컨대 다루는 이론이 LRA 라면, I^T는 선형 부등식의 논리곱이다. 이론 솔버가 I^T를 만족가능하다고 판단하면, F가 만족가능함을 알 수 있고, 끝이다. 그렇지 않으면, DPLL^T 은 I가 모델이 아니라는 사실을 *학습*한다. 그래서 I를 부정한 $\neg I$를 F^B에 논리곱으로 추가한다. 어떤 의미에서 DPLL^T 은 *게으르게(lazily)* 점점 더 많은 사실을 학습해 가며, 추상화를 정제하여, SAT 또는 UNSAT 를 결정할 수 있을 때까지 진행한다.

Example 6.F 다음 LRA 공식 F를 보자:

$$x \geqslant 10 \wedge (x < 0 \vee y \geqslant 0)$$

그 추상화 F^B는

$$p \wedge (q \vee r)$$

이며, 여기서 p는 $x \geqslant 10$, q는 $x < 0$, r은 $y \geqslant 0$을 뜻한다.

첫 번째 반복 DPLL^T 은 F^B에 대해 DPLL 을 호출한다. DPLL 이 부분 모델

$$I_1 = \{p \mapsto \mathsf{true}, q \mapsto \mathsf{true}\}$$

를 반환했다고 하자. I_1을 공식

$$p \wedge q$$

로 표현하자. 이제 I_1이 실제로 F의 모델인지 확인한다. 이를 위해 I_1^T—즉

$$\underbrace{x \geqslant 10}_{p} \wedge \underbrace{x < 0}_{q}$$

—를 이론 솔버에 건넨다. 이론 솔버는 I_1^T이 UNSAT 임을 말할 것이다. x가 동시에 $\geqslant 10$이고 < 0일 수는 없기 때문이다. 따라서 DPLL^T 은 $\neg I_1$을 F^B에 논리곱하여 이 모델을 차단한다. 그러면 CNF 를 유지하면서(왜냐하면 $\neg I_1$은 하나의 절이기 때문이다) F^B는 다음과 같은 공식이 된다:

$$p \wedge (q \vee r) \wedge \underbrace{(\neg p \vee \neg q)}_{\neg I_1}$$

즉, $x \geqslant 10$과 $x < 0$을 동시에 true로 두는 모델은 허용되지 않는다고 말하는 셈이다.

두 번째 반복 두 번째 반복에서, DPLL^T 은 갱신된 F^B에 대해 DPLL 을 호출한다. DPLL 은 더 이상 같은 모델 I_1을 줄 수 없다. 대신 $I_2 = p \wedge \neg q \wedge r$ 같은 모델을 준다고 하자. 이론 솔버는 I_2^T가 만족가능하다고 판단하고, (예: $\{x \mapsto 10,\ y \mapsto 0\}$ 같은) 자신만의 이론 수준 모델을 반환한다. 이제 끝났고, 그 모델을 반환한다.

■

6.4 Tseitin's Transformation

지금까지 공식을 CNF 로 가정해 왔다. 실제로는 아무 공식이나 CNF 로 바꿀 수 있다. 드모르간 법칙(see Chapter 4)을 반복 적용하여 논리합을 논리곱 위로 분배하면 된다. 예컨대 $r \vee (p \wedge q)$는 $(r \vee p) \wedge (r \vee q)$로 동치 변환된다. 안타깝게도 이 변환은 공식의 크기를 지수적으로 폭증시킬 수 있다. 운 좋게도 *체이틴(Tseitin) 변환*으로 알려진 간단한 기법이 있는데, 이는 비-CNF 공식을 크기 선형의 CNF 공식으로 바꿔 준다.

체이틴 변환은 공식 F를 받아 CNF 공식 F'를 만든다. F의 변수 집합은 F' 변수 집합의 부분집합이다; 즉, 체이틴 변환은 새 변수를 만든다. 체이틴 변환은 다음을 보장한다:

1. F'의 임의의 모델은, 새로 추가된 변수들의 해석을 무시하면, F의 모델이기도 하다.
2. 만약 F'가 UNSAT 이라면, F도 UNSAT 이다.

따라서 비-CNF 공식 F가 주어졌을 때, 그 만족가능성을 검사하려면 DPLL 을 F'에 호출하면 된다.

Intuition

체이틴 변환은, 프로그램 안의 복잡한 산술식을 각 명령이 하나의 단항 또는 이항 연산만 적용하는 명령들의 열로 바꾸는 것과 사실상 같다. (이는 고수준 프로그램을 어셈블리나 중간 표현으로 컴파일할 때 컴파일러가 대략 하는 일이다.) 예컨대 다음 함수(Python 문법)를 보자:

```
def f(x,y,z):
    return x + (2*y + 3)
```

return 식은 각 단계가 한두 변수에만 작용하도록 다음처럼 연산들의 열로 다시 쓸 수 있다. 여기서 `t1`, `t2`, `t3`는 임시 변수다:

```
def f(x,y,z):
    t1 = 2 * y
    t2 = t1 + 3
    t3 = x + t2
    return t3
```

직관적으로, 각 부분식을 계산하여 임시 변수에 저장한다: `t1`에는 `2*y`, `t2`에는 `2*y + 3`, `t3`에는 전체 식 `x + (2*y + 3)`이 담긴다.

Tseitin Step 1: NNF

체이틴 변환이 처음 하는 일은 부정을 안쪽으로 밀어 넣어 ¬가 변수 옆에만 나타나도록 하는 것이다. 예: $\neg(p \wedge r)$은 $\neg p \vee \neg r$로 쓴다. 이를 *부정 정규형*(NNF)이라 한다. 다음의 치환 규칙들을 더 이상 적용할 수 없을 때까지 반복 적용하면 어떤 공식이든 NNF로

쉽게 바꿀 수 있다:

$$\neg(F_1 \wedge F_2) \to \neg F_1 \vee \neg F_2$$
$$\neg(F_1 \vee F_2) \to \neg F_1 \wedge \neg F_2$$
$$\neg\neg F \to F$$

즉, $\to$의 왼편 패턴과 일치하는 (부분)공식을 오른편 패턴으로 바꾼다. 이후의 논의에서는 공식이 NNF라고 가정한다.

Tseitin Step 2: Subformula Rewriting

F의 부분공식은, 논리합이나 논리곱을 포함하는 모든 부분공식을 말한다—즉, 리터럴 수준의 부분공식은 고려하지 않는다.

Example 6.G 다음 공식 F는 네 개의 부분공식으로 분해된다:

$$F \triangleq \underbrace{\underbrace{(p \wedge q)}_{F_1} \vee \underbrace{(q \wedge \neg r}_{F_2} \wedge s)}_{F_3}}_{F_4}$$

F_1과 F_2는 가장 *깊은 중첩 수준*에, F_4는 가장 얕은 수준에 있다. F_2는 F_3의 부분공식이며, 모든 F_i는 F_4의 부분공식이다. ■

이제 변환 절차를 보자. F에 부분공식이 n개 있다고 하자:

1. F의 각 부분공식 F_i마다 새로운 변수 t_i를 만든다. *이 변수들은 위의 Python 프로그램에 도입한 임시 변수 t_i에 해당한다.*

2. 다음으로, 가장 깊이 중첩된 부분공식부터 시작하여 각 부분공식 F_i마다 다음 공식을 만든다: F_i가 $\ell_i \circ \ell_i'$ 꼴($\circ \in \{\wedge, \vee\}$, ℓ_i, ℓ_i'는 리터럴)이라고 하자. ℓ_i나 ℓ_i' 중 하나 또는 둘 다가, F_i의 부분공식 F_j를 가리키는 새 변수 t_j일 수도 있다. 다음 공식을 만든다:

   $$F_i' \triangleq t_i \Leftrightarrow (\ell_i \circ \ell_i')$$

   이 공식들은 Python 프로그램에서 임시 변수에 값을 대입하는 것에 해당하며, $\Leftrightarrow$는 변수 대입(=)의 논리적 대응물이다.

Example 6.H 예시를 이어가자. 부분공식 F_1에 대해 다음 공식을 만든다:

$$F_1' \triangleq t_1 \Leftrightarrow (p \wedge q)$$

부분공식 F_2에 대해서는

$$F_2' \triangleq t_2 \Leftrightarrow (q \wedge \neg r)$$

부분공식 F_3에 대해서는

$$F_3' \triangleq t_3 \Leftrightarrow (t_2 \wedge s)$$

(여기서 $q \wedge \neg r$가 변수 t_2로 대체됨을 주목하라.) 마지막으로 부분공식 F_4에 대해

$$F_4' \triangleq t_4 \Leftrightarrow (t_1 \vee t_3)$$

을 만든다. ■

각 F_i'는 CNF 로 쓸 수 있음을 주목하라. 이는 $\Leftrightarrow$의 정의와 드모르간 법칙에 따라 다음이 성립하기 때문이다:

$$\ell_1 \Leftrightarrow (\ell_2 \vee \ell_3) \equiv (\neg\ell_1 \vee \ell_2 \vee \ell_3) \wedge (\ell_1 \vee \neg\ell_2) \wedge (\ell_1 \vee \neg\ell_3)$$

그리고

$$\ell_1 \Leftrightarrow (\ell_2 \wedge \ell_3) \equiv (\neg\ell_1 \vee \ell_2) \wedge (\neg\ell_1 \vee \ell_3) \wedge (\ell_1 \vee \neg\ell_2 \vee \neg\ell_3)$$

마지막으로 다음 CNF 공식을 구성한다:

$$F' \triangleq t_n \wedge \bigwedge_i F_i'$$

구성상, F'의 임의의 모델에서 각 t_i는 부분공식 F_i가 true로 평가될 때 그리고 그럴 때에 한해 true로 할당된다. 따라서 F'의 제약 t_n은 F가 참이어야 함을 말한다. *t_n을, 변환된 Python 프로그램의 return 문이라고 생각해도 된다.*

Example 6.I 예시를 이어가며, 최종적으로 다음 CNF 공식을 구성한다:

$$F' \triangleq t_4 \wedge F_1' \wedge F_2' \wedge F_3' \wedge F_4'$$

위에서 본 바와 같이 모든 F_i'는 CNF 로 쓸 수 있으므로, F'는 CNF 이다. ■

이 시점에서 작업은 끝났다. 어떤 이론의 공식과 이론 솔버가 주어지면, 먼저 체이틴 변환을 이용해 공식을 CNF 로 바꾸고, DPLL^T 을 호출한다.

Looking Ahead

여기서는 DPLL^T 의 요약 버전을 제시했다. 현대 SAT 및 SMT 솔버의 핵심 아이디어 중 하나는 *충돌 주도 절 학습*(conflict-driven clause learning)으로, 만족 대입을 만들지 못하는 해석 집합을 식별하여 탐색 공간을 줄이는 데 도움이 되는 그래프 자료구조다. 관심 있는 독자는 절 학습 및 기타 아이디어에 관한 자세한 설명을 Biere et al. (2009)에서 참고하길 바란다.

또한 널리 쓰이는 SAT 및 SMT 솔버를 직접 다뤄 보길 권한다. 예를 들어 MiniSAT (Een, 2005)은 이름에서 알 수 있듯 코드베이스가 작고 읽기 쉽다. SMT 솔버로는 Z3 (de Moura and Bjørner, 2008) 와 CVC4 (Barrett et al., 2011)를 추천한다. SMT 솔버의 흥미로운 기반 아이디어 중 하나는 *이론 결합*(theory combination) (Nelson and Oppen, 1979)으로, 서로 다른 이론들을 결합한 공식을 풀 수 있게 해 준다. 이는 문자열, 배열, 정수 등 다양한 자료를 다루는 일반 프로그램 검증에서 유용하다. 미래에는 신경망을 더 큰 소프트웨어의 구성 요소로 보기 시작할 것이므로, 신경망 검증에서도 이론 결합이 필요해질 것이라고 강하게 예상한다.

Chapter 7

Neural Theory Solvers

앞 장에서는 FOL 의 공식을 푸는 방법으로 DPLL^T 를 논의했다. 이 장에서는 선형 실수 산술에서 리터럴의 논리곱을 푸는 *심플렉스(Simplex) 알고리즘*을 살펴본다. 그 다음, 보통은 분기(논리합)로 부호화하여 SAT 솔버가 다루게 되는 정류 선형 단위(ReLU)를, 솔버가 *자체적으로* 처리하도록 확장한다.

7.1 Theory Solving and Normal Forms

The Problem

LRA 의 이론 솔버는 다음과 같은 선형 부등식들의 논리곱 형태의 공식 F를 입력으로 받는다:

$$\bigwedge_{i=1}^{n} \left(\sum_{j=1}^{m} c_{ij} \cdot x_j \geqslant b_i \right)$$

여기서 $c_{ij}, b_i \in \mathbb{R}$이다. 목표는 F의 만족가능성을 검사하고, 만족가능하다면 모델 $I \models F$를 찾는 것이다.

여기서 우리는 엄격 부등식($>$)을 사용하지 않는다. 아래의 방법은 엄격 부등식으로도 쉽게 일반화되지만, 단순화를 위해 비엄격 부등식만을 사용한다.[1]

[1] 엄격 부등식을 다루는 방법은 Dutertre and De Moura (2006) 를 보라. 신경망의 성질을 검증하는 대부분의 경우, 네트워크 의미나 성질을 부호화하는 데 엄격 부등식이 필요하지 않다.

Simplex Form

다음 절에서 다룰 심플렉스 알고리즘은 (마치 DPLL 이 명제 공식을 CNF 로 기대하듯) 입력 공식이 특정 꼴이기를 기대한다. 구체적으로, 심플렉스는 다음 두 형태의 제약들의 논리곱을 기대한다:

$$\sum_i c_i \cdot x_i = 0$$

(같음식, equality)과

$$l_i \leqslant x_i \leqslant u_i$$

(변수 경계, bounds)이다. 여기서 $u_i, l_i \in \mathbb{R} \cup \{\infty, -\infty\}$이다. ∞(resp. $-\infty$)는 변수의 상한(resp. 하한)이 없음을 나타낸다.

따라서 주어진 공식 F를 위의 형태(이른바 *심플렉스 형식*, 또는 *슬랙 형식*)의 동치 공식으로 변환해야 한다. 이 변환은 꽤 간단하다.

$$F \triangleq \bigwedge_{i=1}^{n} \left(\sum_{j=1}^{m} c_{ij} \cdot x_j \geqslant b_i \right)$$

라고 하면, 각 부등식으로부터 하나의 같음식과 하나의 경계를 만든다. i번째 부등식

$$\sum_{j=1}^{m} c_{ij} \cdot x_j \geqslant b_i$$

으로부터 같음식

$$s_i = \sum_{j=1}^{m} c_{ij} \cdot x_j$$

과 경계

$$s_i \geqslant b_i$$

를 만든다. 여기서 s_i는 *슬랙 변수(slack variable)*라 부르는 새 변수다. 슬랙 변수는 체이틴 변환(Chapter 6)에서 도입한 임시 변수와 유사한 역할을 한다.

Example 7.A 다음 공식을 (이 장 전체에 걸친) 실행 예제로 사용하자:

$$\begin{aligned} x + y &\geqslant 0 \\ -2x + y &\geqslant 2 \\ -10x + y &\geqslant -5 \end{aligned}$$

가독성을 위해 논리곱 기호는 생략하고 부등식만 나열한다. 이를 심플렉스 형식 F_s로 바꾸면:

$$
\begin{aligned}
s_1 &= x + y \\
s_2 &= -2x + y \\
s_3 &= -10x + y \\
s_1 &\geqslant 0 \\
s_2 &\geqslant 2 \\
s_3 &\geqslant -5
\end{aligned}
$$

■

이 변환은 원래 공식을 만족가능성 보존 방식으로 다시 쓴 것에 불과하다. F의 심플렉스 형식을 F_s라고 하자. 그러면 (비-CNF 공식을 위한 체이틴 변환의 아날로그에 해당하는) 다음이 보장된다:

1. F_s의 임의의 모델은, 슬랙 변수에 대한 할당을 무시하면, F의 모델이다.
2. F_s가 UNSAT 이면, F도 UNSAT 이다.

7.2 The Simplex Algorithm

이제 심플렉스 알고리즘을 소개할 준비가 되었다. 이 알고리즘은 1947년 조지 댄치그(George Dantzig)가 개발한 아주 오래된 아이디어이다(기원에 관한 회고는 (Dantzig, 1990) 참조). 알고리즘의 원래 목표는 어떤 목적함수를 최대로 만드는 만족 해를 찾는 것이다. 검증에서는 보통 *아무* 만족 해나 찾으면 충분하므로, 여기서는 심플렉스의 부분집합에 해당하는 알고리즘을 제시한다.

Intuition

심플렉스를, 모델을 찾는 동시에 UNSAT 증명을 찾는 절차로 생각할 수 있다. 일부 해석에서 출발하여, 매 반복마다 그것을 갱신하면서 모델을 찾거나 UNSAT 증명을 발견할 때까지 진행한다. 시작 해석 I는 모든 변수를 0으로 둔다. 이 할당은 같음식은 모두 만족시키지만, 경계는 만족시키지 못할 수 있다. 매 반복에서 심플렉스는 만족되지 않은

경계를 하나 골라 그것을 만족시키도록 I를 수정하거나, 공식이 UNSAT 임을 밝혀낸다. 수학으로 들어가기 전에, 만족가능한 예를 그림으로 보자.

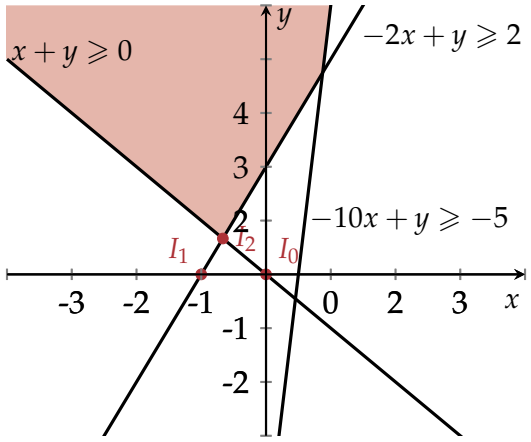


Figure 7.1 심플렉스 예시

Example 7.B Figure 7.1 에 도시된 실행 예제를 다시 보자. 음영으로 칠해진 영역이 만족 해들의 집합이다. 각 부등식은 *반공간*을 정의하며, 즉 $\mathbb{R}^2$를 둘로 가른다. 모든 부등식을 함께 고려하면 (모든 반공간의 교집합인) 음영 영역이 된다.

$$x + y \geqslant 0$$
$$-2x + y \geqslant 2$$
$$-10x + y \geqslant -5$$

심플렉스는 초기 해석

$$I_0 = \{x \mapsto 0, y \mapsto 0\}$$

에서 시작하며(그림의 I_0), 이는 공식을 만족하지 않는다.

심플렉스는

$$I_0 \not\models -2x + y \geqslant 2$$

임을 보고 x를 -1로 낮춰 I_1을 만든다. 이어서

$$I_1 \not\models x + y \geqslant 0$$

임을 보고 y를 0에서 2/3으로 올려 만족 해 I_2에 도달한다. (이때 x도 I_2에서 변하는데, 그 이유는 곧 본다.) 심플렉스는 일종의 두더지 잡기(Whac-A-Mole)를 하듯 어떤 부등식을 만족시키려다 다른 부등식을 깨뜨리기를 반복하며, 결국 모든 부등식을 만족하는 할당에 도달한다. 다행히 알고리즘은 실제로 종료한다. ■

Basic and Non-basic Variables

심플렉스는 입력이 심플렉스 형식임을 가정한다. 공식의 변수들은 두 부류로 나뉜다:

기저 변수(basic variables) 같음식의 좌변에 등장하는 변수들; 초기에는 슬랙 변수들이 기저 변수다.

비기저 변수(non-basic variables) 나머지 모든 변수들.

심플렉스가 진행되면서 공식을 다시 쓰기 때문에, 어떤 기저 변수는 비기저가 되고 그 반대도 생긴다.

Example 7.C 실행 예제에서, 초기 기저 변수 집합은 $\{s_1, s_2, s_3\}$이고 비기저 변수는 $\{x, y\}$이다. ■

심플렉스의 종료성을 보장하기 위해 모든(기저 및 비기저) 변수에 대해 전순서를 고정한다. 따라서 "*첫 번째 변수*"라고 하면 이 순서에서의 첫 변수를 뜻한다. 표기를 간단히 하기 위해 변수들을 $x_1, \ldots, x_n$ 꼴이라 하자. 기저 변수 x_i와 비기저 변수 x_j에 대해, 다음 같음식에서 x_j의 계수를 c_{ij}로 표기한다:

$$x_i = \ldots + c_{ij} \cdot x_j + \ldots$$

또한 변수 x_i의 하한과 상한을 각각 l_i, u_i로 쓴다. 상한(resp. 하한)이 없으면 ∞(resp. $-\infty$)로 둔다. 비슬랙 변수에는 경계가 없음을 주의하라.

Simplex in Detail

이제 Algorithm 3 의 심플렉스 알고리즘을 제시한다. 알고리즘은 다음 두 불변식을 유지한다:

1. 해석 I는 항상 같음식을 만족한다(따라서 위배될 수 있는 것은 경계뿐이다). 초기에는 모든 변수가 0이므로 자명하다.

2. 비기저 변수들의 경계는 항상 만족된다. 초기에는 비기저 변수에 경계가 없으므로 자명하다.

루프의 각 반복에서, 심플렉스는 현재 해석에서 경계를 만족하지 않는 기저 변수를 찾고, 해석을 고치려 시도한다. $x_i < l_i$ 또는 $x_i > u_i$의 두 대칭적인 경우가 있고, 이는 *if* 문의 두 분기로 표현된다.

먼저 $x_i < l_i$의 경우를 보자. x_i가 하한보다 작으므로 I에서 그 값을 키워야 한다. 이를 비기저 변수 x_j의 값을 바꾸어 *간접적으로* 한다. 그런데 어떤 x_j를 골라야 할까? 원칙적으로는 계수 $c_{ij} \neq 0$인 아무 x_j나 골라 그 값을 조정할 수 있다. 알고리즘을 보면 몇 가지 추가 조건이 있다. 이 조건을 만족하는 x_j를 찾지 못하면 문제는 UNSAT 이다. (UNSAT 판정 조건은 곧 설명한다.) 일단 그런 x_j를 찾았다고 하자. x_j의 현재 값을 $\frac{l_i - I(x_i)}{c_{ij}}$만큼 증가시키면, x_i의 값은 정확히 $l_i - I(x_i)$만큼 증가하여 딱 하한을 만족하게 된다, 즉 $I(x_i) = l_i$. 비기저 변수를 바꾸면 기저 변수의 값이 자동으로 (같음식을 통해) 바뀐다고 가정한다. 이로써 첫 번째 불변식이 유지된다.[2]

x_j를 갱신하고 나면 x_j의 경계를 위배했을 수도 있다. 따라서 공식을 다시 써서 x_j를 기저 변수로, x_i를 비기저 변수로 바꾼다. 이를 *피벗(pivot)*이라 하며, 기계적으로 다음과 같이 수행한다: 비기저 변수들의 인덱스 집합을 N이라 할 때

$$x_i = \sum_{k \in N} c_{ik} x_k$$

를 잡아 x_j를 좌변으로 옮겨 다시 쓴다:

$$x_j = \underbrace{-\frac{x_i}{c_{ij}} + \sum_{k \in N \setminus \{j\}} \frac{c_{ik}}{c_{ij}} x_k}_{x_j\text{를 이 식으로 치환}}$$

이제 다른 모든 같음식에서 x_j를 위 식으로 치환한다. 그러면 x_j는 좌변에 한 번만 등장하는 같음식이 되고, 피벗 이후 x_j가 기저 변수, x_i가 비기저 변수가 된다.

[2] 기저 변수를 *종속 변수(dependent)*로, 비기저 변수를 *독립 변수(independent)*로 부르기도 한다. 이는 기저 변수의 값이 비기저 변수의 값에 의존함을 뜻한다.

Algorithm 3: Simplex

Data: 심플렉스 형식의 공식 F
Result: $I \models F$ 또는 UNSAT

I를 $fv(F)$의 모든 변수를 0으로 두는 해석이라 하자
while *true* **do**
  if $I \models F$ **then return** I
  $I(x_i) < l_i$ 또는 $I(x_i) > u_i$인 *첫 번째* 기저 변수 x_i를 잡는다
  if $I(x_i) < l_i$ **then**
    다음 조건을 만족하는 *첫 번째* 비기저 변수 x_j를 잡는다:

$$(I(x_j) < u_j \text{ and } c_{ij} > 0) \text{ or } (I(x_j) > l_j \text{ and } c_{ij} < 0)$$

    if *그런 x_j가 없다* **then return** UNSAT
    $I(x_j) \leftarrow I(x_j) + \frac{l_i - I(x_i)}{c_{ij}}$
  else
    다음 조건을 만족하는 *첫 번째* 비기저 변수 x_j를 잡는다:

$$(I(x_j) > l_j \text{ and } c_{ij} > 0) \text{ or } (I(x_j) < u_j \text{ and } c_{ij} < 0)$$

    if *그런 x_j가 없다* **then return** UNSAT
    $I(x_j) \leftarrow I(x_j) + \frac{u_i - I(x_i)}{c_{ij}}$
  x_i와 x_j를 피벗한다

Example 7.D 이제 실행 예제를 자세히 따라가 보자. 공식은 다음과 같다:

$$\begin{aligned} s_1 &= x + y \\ s_2 &= -2x + y \\ s_3 &= -10x + y \\ s_1 &\geqslant 0 \\ s_2 &\geqslant 2 \\ s_3 &\geqslant -5 \end{aligned}$$

변수 순서를 다음과 같이 두자:

$$x, y, s_1, s_2, s_3$$

초기에는 s_1과 s_3의 경계는 만족되지만, s_2는 $s_2 \geqslant 2$인데 $I_0(s_2) = 0$이므로 위배된다 (초기에는 모두 0이기 때문).

첫 번째 반복 첫 반복에서, 순서상 먼저인 변수 x를 골라 s_2의 경계를 고친다. x는 무경계 변수(하한 $-\infty$, 상한 ∞)이므로 조건을 쉽게 만족한다. s_2를 2까지 올리기 위해 $I_0(x)$를 -1로 내리면, 다음과 같은 만족 해를 얻는다:

$$I_1 = \{x \mapsto -1,\ y \mapsto 0,\ s_1 \mapsto -1,\ s_2 \mapsto 2,\ s_3 \mapsto 10\}$$

(그림 Figure 7.1 을 다시 보라.) 이제 s_2와 x를 피벗하면 다음 같음식을 얻는다(경계는 변하지 않는다):

$$\begin{aligned} x &= 0.5y - 0.5s_2 \\ s_1 &= 1.5y - 0.5s_2 \\ s_3 &= -4y + 5s_2 \end{aligned}$$

두 번째 반복 이제 경계를 위배하는 유일한 기저 변수는 s_1이다($I_1(s_1) = -1 < 0$). 첫 번째로 조정 가능한 비기저 변수는 y다. $I(y)$를 $1/1.5 = 2/3$만큼 올리면 다음 해석을 얻는다:

$$I_2 = \{x \mapsto -2/3,\ y \mapsto 2/3,\ s_1 \mapsto 0,\ s_2 \mapsto 2,\ s_3 \mapsto 7/3\}$$

이때 y와 s_1을 피벗한다.

세 번째 반복 $I_2 \models F$이므로 심플렉스는 종료한다.

■

Why is Simplex Correct?

먼저, 심플렉스가 왜 종료하는가? 그 답은 변수에 순서를 두고 항상 *첫 번째*로 경계를 위배한 변수를 고른다는 점 때문이다. 이는 *블랜드의 규칙(Bland's rule)*로 알려져 있다 (Bland, 1977). 블랜드의 규칙은 같은 기저/비기저 변수 집합을 다시 방문하지 않음을 보장한다.

둘째, 심플렉스는 정말 정확한가? 심플렉스가 해석 I를 반환하면 종료 직전에 $I \models F$를 확인하므로 자명하다. 그렇다면 UNSAT 을 말하는 경우는? 예제로 설명한다.

Example 7.E 다음 심플렉스 형식을 보자:

$$
\begin{aligned}
s_1 &= x + y \\
s_2 &= -x - 2y \\
s_3 &= -x + y \\
s_1 &\geqslant 0 \\
s_2 &\geqslant 2 \\
s_3 &\geqslant 1
\end{aligned}
$$

이 공식은 UNSAT — 좋아하는 SMT 솔버로 확인해 보라. 심플렉스가 (1) s_1과 x, (2) s_2와 y를 차례로 피벗했다고 하자.

첫 피벗 뒤 공식:

$$
\begin{aligned}
x &= s_1 - y \\
s_2 &= -s_1 - y \\
s_3 &= -s_1 + 2y
\end{aligned}
$$

둘째 피벗 뒤 공식:

$$
\begin{aligned}
x &= 2s_1 + s_2 \\
y &= -s_2 - s_1 \\
s_3 &= -3s_1 - 2s_s
\end{aligned}
$$

알고리즘은 모든 비기저 변수가 경계를 만족한다는 불변식을 유지한다. 즉 $s_1 \geqslant 0$이고 $s_2 \geqslant 2$다. 이제 s_3가 경계를 위배한다고 하자, 즉

$$-3s_1 - 2s_2 < 1$$

이를 고치는 유일한 방법은 s_1과 s_2를 줄이는 것이다. 하지만 s_1을 0, s_2를 2(각각의 하한)로 둔다 해도 $s_3 \geqslant 1$을 만들 수 없다. *모순!* 따라서 심플렉스는 공식을 UNSAT 로 판단한다. Algorithm 3 의 x_j 선택 조건은 이 논증을 부호화한다. ■

위 예가 보여주듯, 심플렉스는 선형 부등식 집합이 UNSAT 임을 보여 주는 모순 증명을 구성한다고 볼 수 있다.

7.3 The Reluplex Algorithm

심플렉스를 DPLL^T 의 이론 솔버로 사용하면 LRA 공식을 풀 수 있다. 즉, 지금까지의 전개로는 ReLU 같은 조각별 선형 활성화를 가진 신경망을 알고리즘적으로 다룰 수 있다. 불행히도 이 접근은 큰 네트워크로는 확장이 안 되는 것으로 드러났다. 그 이유 중 하나는 ReLU가 Chapter 5 에서 본 대로 논리합으로 부호화된다는 점이다. 이 말은 DPLL^T 의 SAT-풀기 부분이 이 논리합을 다루며, ReLU가 활성(출력=입력)인지 비활성(출력=0)인지의 모든 경우를 고려하느라 ReLU 수에 지수적인 횟수로 심플렉스를 호출할 수 있음을 뜻한다.

이 문제를 해결하기 위해 Katz et al. (2017) 는 선형 부등식에 더해 ReLU 제약까지 *자체적으로* 다루는 심플렉스의 확장판인 *Reluplex*를 제안했다. 핵심 아이디어는 ReLU에 대한 경우 분기를 *되도록 늦추는* 것이다. 최악의 경우 Reluplex도 DPLL^T +심플렉스처럼 지수 폭발을 겪을 수 있지만, 경험적으로는 더 큰 신경망으로 SMT 풀기를 스케일시키는 유망한 접근으로 보였다. 아래에서 Reluplex 알고리즘을 제시한다.

Reluplex Form

심플렉스와 마찬가지로, Reluplex도 입력 공식의 형식을 기대한다. 이를 *Reluplex 형식*이라 부르며, (1) 같음식(심플렉스와 동일), (2) 경계(심플렉스와 동일), (3) 다음 꼴의 *ReLU 제약*을 포함한다:

$$x_i = \text{relu}(x_j)$$

부등식과 ReLU 제약의 논리곱이 주어지면, 부등식을 심플렉스 형식으로 바꾸어 Reluplex 형식으로 만들 수 있다. 추가로 각 ReLU 제약 $x_i = \text{relu}(x_j)$에 대해, 정의상 $x_i \geqslant 0$ 이므로 이 경계를 더할 수 있다.

Example 7.F 다음 공식을 보자:

$$x + y \geqslant 2$$
$$y = \text{relu}(x)$$

이를 다음의 Reluplex 형식으로 변환한다:

$$
\begin{aligned}
s_1 &= x + y \\
y &= \text{relu}(x) \\
s_1 &\geqslant 2 \\
y &\geqslant 0
\end{aligned}
$$

■

Reluplex in Detail

Reluplex 알고리즘을 설명한다. Katz et al. (2017) 의 원래 발표는 정답에 도달할 때까지 비결정적으로 적용할 수 있는 규칙 집합 형태이다. 여기서는 구체적인 스케줄을 제시한다.

Reluplex의 핵심 아이디어는 같음식과 경계에 대해 심플렉스를 호출하고, 심플렉스가 준 해석을 ReLU 제약까지 만족하도록 *손봐* 주는 것이다. 알고리즘은 Algorithm 4에 나와 있다.

먼저, 원래 공식 F에서 ReLU 제약을 제거한 F'에 대해 심플렉스를 호출한다. 심플렉스가 UNSAT 을 반환하면 $F \Rightarrow F'$가 유효하므로 F도 UNSAT 임을 안다. 그렇지 않고 심플렉스가 모델 $I \models F'$를 반환하더라도 F'가 더 약한(제약이 덜한) 공식이므로 $I \models F$가 아닐 수 있다.

$I \not\models F$이면, 어떤 ReLU 제약이 위배되었다는 뜻이다. 위배된 ReLU 제약 $x_i = \text{relu}(x_j)$를 하나 골라 그 제약을 위배하지 않게 I를 수정한다. 이때 x_i나 x_j 중 기저 변수가 있으면 비기저 변수와 피벗한다. 이는 x_i 또는 x_j의 값을 바꾸고 싶은데, $c_{ij} \neq 0$이고 둘 중 하나가 기저 변수이면 다른 한쪽 값까지 영향을 줄 수 있기 때문이다.[3] 마지막으로 x_i나 x_j의 값을 수정하여 $I \models x_i = \text{relu}(x_j)$가 되게 한다. 둘 중 무엇을 고칠지는 구현이 선택한다.

문제는 ReLU 제약을 고치는 과정에서 경계를 위배할 수 있고, 그러면 심플렉스를 다시 호출해야 한다는 점이다. Reluplex의 해석 I는 심플렉스 호출이 수정하는 것과 같은 객체라고 가정한다.

[3] 이 조건들은 Katz et al. (2017) 에는 명시적이지 않지만, 없으면 ReLU 제약 위배를 고치지 못한 채 (혹은 Katz et al. (2017) 의 Update 규칙을) 헛반복할 수 있다.

Algorithm 4: Reluplex

Data: Reluplex 형식의 공식 F
Result: $I \models F$ 또는 UNSAT

I를 $fv(F)$의 모든 변수를 0으로 두는 해석이라 하자
F'를 F에서 ReLU 제약을 뺀 비-ReLU 제약이라 하자
while *true* **do**
  ▷ 심플렉스 호출(초기 해석에 대한 참조를 넘겨 심플렉스가 그것을 수정하도록 한다)
  $r \leftarrow \text{Simplex}(F', I)$
  If r가 UNSAT **then return** UNSAT
  r는 해석 I이다
  if $I \models F$ **then return** I

  ▷ 위배된 ReLU 제약 처리
  $I(x_i) \neq \text{relu}(I(x_j))$인 ReLU 제약 $x_i = \text{relu}(x_j)$를 하나 잡는다
  if x_i*가 기저 변수* **then**
    $k \neq j, c_{ik} \neq 0$인 비기저 변수 x_k와 x_i를 피벗
  if x_j*가 기저 변수* **then**
    $k \neq i, c_{jk} \neq 0$인 비기저 변수 x_k와 x_j를 피벗
  다음 중 하나를 수행한다:

$$I(x_i) \leftarrow \text{relu}(I(x_j)) \quad \textbf{or} \quad I(x_j) \leftarrow I(x_i)$$

  ▷ 케이스 분기(종료성 보장)
  if $u_j > 0, l_j < 0$, *그리고* $x_i = relu(x_j)$*를* τ*회 초과 시도했다면* **then**
    $r_1 \leftarrow \text{Reluplex}(F \wedge x_j \geqslant 0 \wedge x_i = x_j)$
    $r_2 \leftarrow \text{Reluplex}(F \wedge x_j \leqslant 0 \wedge x_i = 0)$
    if $r_1 = r_2 =$ UNSAT **then return** UNSAT
    if $r_1 \neq$ UNSAT **then return** r_1
    return r_2

Case Splitting

마지막 단계(케이스 분기) 없이 Reluplex를 적용하면 종료하지 않을 수 있다. 구체적으로, 심플렉스가 모든 경계를 만족시키지만 ReLU를 위배하고, ReLU를 만족시키면 경계를 위배하는 일이 반복될 수 있다.

Reluplex의 마지막 단계는 무한 루프에 빠지지 않도록, 특정 ReLU 제약을 τ회 이상 고치려 들지 못하게 한다(어떤 고정된 임계치). 임계치를 넘으면 ReLU 제약 $x_i = \text{relu}(x_j)$를 두 경우로 분기한다:

$$F_1 \triangleq x_j \geqslant 0 \wedge x_i = x_j$$

그리고

$$F_2 \triangleq x_j \leqslant 0 \wedge x_i = 0$$

Reluplex는 두 문제 인스턴스 $F \wedge F_1$과 $F \wedge F_2$를 재귀적으로 푼다. 둘 다 UNSAT 이면 F는 UNSAT, 둘 중 하나라도 SAT면 F는 SAT이다. 이는

$$F \equiv (F \wedge F_1) \vee (F \wedge F_2)$$

이기 때문이다.

Looking Ahead

제약 기반 검증에 관한 논의를 마친다. 다음 파트에서는, 어떤 경우에는 증명을 제공하지 못하는 대가로 더 효율적인 다양한 접근을 살펴본다.

여기서 다루지 못한 흥미로운 문제들이 여럿 있다. 그 중 핵심은 기계 산술에 대한 건전성(soundness)이다. 우리는 실수 값을 가정했지만, 현실에서는 기계 산술을 사용한다. 최근 연구는 LRA 에서 검증된 신경망도 비트 수준의 동작을 고려하면 실제로는 견고하지 않을 수 있음을 보였다 (Jia and Rinard, 2020b).

분석의 확장성 문제도 있다. 가변 정밀 유리수 산술은 피벗 연산 때문에 분자/분모 크기가 폭증하여 매우 비쌀 수 있다. Reluplex (Katz et al., 2017) 는 부동소수 근사를 사용하고, 반올림 오차를 추적하여 결과의 건전성을 신중히 보장한다.

이 글을 쓰는 시점에, 제약 기반 검증 기법들은 대략 수십만 개 ReLU를 가진 신경망 정도까지 스케일했다 (Tjeng et al., 2019a). 최신 신경망에 비하면 작은 편이다. 그럼에도 검증 문제가 NP-난해함을 감안하면 (Katz et al., 2017) 큰 성취다. 하지만 이 기술을 더 얼마나 밀어붙일 수 있을지는 불분명하다. 제약 기반 검증을 더 키우려면 두 축에서의 진전이 필요하다: (1) 검증 친화적으로(더 적은 ReLU가 좋다) 설계/학습된 신경망 개발, (2) SMT 솔버 및 MILP 솔버를 떠받치는 알고리즘의 발전.

Part III

Abstraction-Based Verification

Bibliography

Martín Abadi and Gordon D. Plotkin. A simple differentiable programming language. *Proc. ACM Program. Lang.*, 4(POPL):38:1–38:28, 2020. doi: 10.1145/3371106. URL https://doi.org/10.1145/3371106.

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Turing Alan. On checking a large routine. In *Report of a Conference on 11i9h Speed Automatic Calculating Machines*, pages 67–69, 1949.

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi: 10.1007/978-3-642-22110-1_14. URL https://doi.org/10.1007/978-3-642-22110-1_14.

Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2613–2621, 2016. URL https://proceedings.neurips.cc/paper/2016/hash/980ecd059122ce2e50136bda65c25e07-Abstract.html.

Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009. ISBN 978-1-58603-929-5.

Robert G. Bland. New finite pivoting rules for the simplex method. *Math. Oper. Res.*, 2(2):103–107, 1977. doi: 10.1287/moor.2.2.103. URL https://doi.org/10.1287/moor.2.2.103.

Bob F Caviness and Jeremy R Johnson. *Quantifier elimination and cylindrical algebraic decomposition*. Springer Science & Business Media, 2012.

Alonzo Church. A note on the entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936. doi: 10.2307/2269326. URL https://doi.org/10.2307/2269326.

George B Dantzig. Origins of the simplex method. In *A history of scientific computing*, pages 141–151. 1990.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24.

Bruno Dutertre and Leonardo De Moura. Integrating simplex with dpll (t). *Computer Science Laboratory, SRI International, Tech. Rep. SRI-CSL-06-01*, 2006.

Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. Hotflip: White-box adversarial examples for text classification. In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 2: Short Papers*, pages 31–36. Association for Computational Linguistics, 2018. doi: 10.18653/v1/P18-2006. URL https://www.aclweb.org/anthology/P18-2006/.

Niklas Een. Minisat: A sat solver with conflict-clause minimization. In *Proc. SAT-05: 8th Int. Conf. on Theory and Applications of Satisfiability Testing*, pages 502–518, 2005.

Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 269–286. Springer, 2017. doi: 10.1007/978-3-319-68167-2_19. URL https://doi.org/10.1007/978-3-319-68167-2_19.

Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 1625–1634. IEEE Computer Society, 2018. doi: 10.1109/CVPR.2018.00175. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Eykholt_Robust_Physical-World_Attacks_CVPR_2018_paper.html.

Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN 978-0-262-03561-3. URL http://www.deeplearningbook.org/.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366,

1989. doi: 10.1016/0893-6080(89)90020-8. URL https://doi.org/10.1016/0893-6080(89)90020-8.

Po-Sen Huang, Robert Stanforth, Johannes Welbl, Chris Dyer, Dani Yogatama, Sven Gowal, Krishnamurthy Dvijotham, and Pushmeet Kohli. Achieving verified robustness to symbol substitutions via interval bound propagation. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 4081–4091. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1419. URL https://doi.org/10.18653/v1/D19-1419.

Kai Jia and Martin Rinard. Efficient exact verification of binarized neural networks, 2020a. URL https://proceedings.neurips.cc/paper/2020/hash/1385974ed5904a438616ff7bdb3f7439-Abstract.html.

Kai Jia and Martin Rinard. Exploiting verified neural networks via floating point numerical error. *CoRR*, abs/2003.03021, 2020b. URL https://arxiv.org/abs/2003.03021.

Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017. doi: 10.1007/978-3-319-63387-9_5. URL https://doi.org/10.1007/978-3-319-63387-9_5.

Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993. doi: 10.1016/

S0893-6080(05)80131-5. URL https://doi.org/10.1016/S0893-6080(05)80131-5.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 807–814. Omnipress, 2010. URL https://icml.cc/Conferences/2010/papers/432.pdf.

Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. Verifying properties of binarized deep neural networks. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 6615–6624. AAAI Press, 2018. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16898.

Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979. doi: 10.1145/357073.357079. URL https://doi.org/10.1145/357073.357079.

Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2018. URL http://neuralnetworksanddeeplearning.com/.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Luca Pulina and Armando Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 243–257. Springer, 2010. doi: 10.1007/978-3-642-14295-6_24. URL https://doi.org/10.1007/978-3-642-14295-6_24.

Chongli Qin, Krishnamurthy (Dj) Dvijotham, Brendan O'Donoghue, Rudy Bunel, Robert Stanforth, Sven Gowal, Jonathan Uesato, Grzegorz Swirszcz, and Pushmeet Kohli. Verification of non-linear specifications for neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=HyeFAsRctQ.

Benjamin Sherman, Jesse Michel, and Michael Carbin. λ_s: computable semantics for differentiable programming with higher-order functions and datatypes. *Proc. ACM Program. Lang.*, 5(POPL):1–31, 2021. doi: 10.1145/3434284. URL https://doi.org/10.1145/3434284.

Aishwarya Sivaraman, Golnoosh Farnadi, Todd D. Millstein, and Guy Van den Broeck. Counterexample-guided learning of monotonic neural networks. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/8ab70731b1553f17c11a3bbc87e0b605-Abstract.html.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014. URL http://arxiv.org/abs/1312.6199.

Alfred Tarski. A decision method for elementary algebra and geometry. In *Quantifier elimination and cylindrical algebraic decomposition*, pages 24–84. Springer, 1998.

Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019a. URL https://openreview.net/forum?id=HyGIdiRqtm.

Vincent Tjeng, Kai Yuanqing Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019b. URL https://openreview.net/forum?id=HyGIdiRqtm.

Alan Turing. Intelligent machinery. 1948. *The Essential Turing*, page 395, 1969.