6. Equational Logic

등식 논리(Equational Logic)의 목적

- 목표: 어떤 두 식 t, u가 명세 E 안에서 **논리적으로 동등한가(logically equivalent)**를 판단하는 것.
- 즉, t = u가 E의 **방정식들로부터 논리적으로 따라오는가(follows logically)**를 논한다.

논리적 동등성의 두 가지 개념

1. **Signature(연산자 체계)를 고려하지 않는 경우**
   - t = u가 단순히 E 안의 등식들로부터 유도될 수 있으면 성립.
   - 단순한 수학적 추론 수준.
2. **Signature를 고려하는 경우**
   - 모든 ground substitution σ에 대해 tσ = uσ가 성립해야 함.
   - 즉, 모든 구체적인 값 대입에서도 두 식이 동일해야 함.

예시: 군(Group) 이론

- 군 공리(axioms):
  $\{e \circ x = x,\ i(x) \circ x = e,\ (x \circ y) \circ z = x \circ (y \circ z)\}$
- 예를 들어 $x \circ i(x) = e$가 이 공리들로부터 논리적으로 따라오는지를 본다.
  → "모든 군 구조(all groups)"에서 이 등식이 참이라면, 논리적으로 따라온다고 말한다.

문제점과 해결책

- 문제: "모든 가능한 구조(all possible structures)"를 실제로 전부 확인하는 건 불가능하다.
- 해결책:
  → **Equational Logic**을 사용해서 이 추론을 **형식적으로(reasoning)** 수행한다.
  즉, 모든 구조를 검사하지 않고도 t = u의 타당성을 증명할 수 있게 한다.

'의도된 구조(intended structure)' 개념

- 예: 자연수 덧셈 시스템 **NAT-ADD**
  - 등식: 0 + M = M, s(M) + N = s(M + N)
  - 우리는 "모든 가능한 구조"에서 이 등식이 참인지에는 관심이 없다.
    → **오직 자연수 구조(natural numbers)**에서만 성립하는지 확인하고 싶다.
- 예를 들어, M + N = N + M (교환법칙)은
  - 모든 구조에서는 성립하지 않지만,
  - 의도된 자연수 구조에서는 성립한다.

귀납적 정리(Inductive Theorem)

- "의도된 구조"에서만 성립하는 등식들을 **귀납적 정리(inductive theorem)**라고 부른다.

- 예: m + n = n + m은 NAT-ADD에서 귀납적 정리이다.
- Chapter 7에서는 "의도된 모델(intended model)"의 정의와 "귀납적 정리가 그 모델에서 성립한다"는 의미를 형식적으로 다룬다.

6.1–6.2절 미리보기

- **6.1:** 등식 논리의 기본 개념(Formal rules of equational logic)
- **6.2:** 귀납적 정리를 증명하는 방법(Proving inductive theorems)
- 가정: (Σ,E)는 비정렬 명세(unsorted specification)이고, 조건부 방정식이 없으며, Σ에는 최소 하나의 상수가 존재한다.

6.1 Equational Logic

Equational Logic의 기본 개념

- **목표:** 어떤 두 식 `t`와 `u`가 "논리적으로 같다(=)"고 말할 수 있는가를 정의함.
- **기호:**
  `E ⊢ t = u`
  → "등식 집합 E로부터 `t = u`를 증명(prove)할 수 있다."
- 즉, E 안의 규칙들로부터 `t`를 `u`로 바꿀 수 있음을 의미한다.

Equational Logic의 다섯 가지 규칙

규칙	이름	설명
E_1	**Substitutivity**	E 안의 식 `l = r`에 어떤 치환 σ를 적용해도 `lσ = rσ`가 성립함
E_2	**Reflexivity**	`t = t` (모든 식은 자기 자신과 같다)
E_3	**Symmetry**	`t = u`라면 `u = t`도 참
E_4	**Transitivity**	$t_1 = t_2$이고 $t_2 = t_3$라면 $t_1 = t_3$
E_5	**Congruence**	함수 내부에서도 성립 ($f(t_1,\ldots,tn) = f(u_1,\ldots,un)$)

이 다섯 개로 모든 등식 추론이 이루어진다.

예시 (Example 6.1)

E = f(x) = g(x), a = b, g(c) = c

- `E ⊢ b = a`는 이렇게 증명 가능:
  1. `a = b` (E_1 적용)
  2. 대칭(Symmetry) → `b = a`

     → 자명한 등식임을 논리적으로 확인한 것.
- 그다음 `E ⊢ f(a) = g(b)`를 증명하면:
  - Substitutivity, Congruence, Transitivity 규칙을 순서대로 써서 증명 가능.
  - 마지막엔 "proof tree" 형태로 한눈에 정리 가능.

$$\dfrac{\dfrac{\dfrac{}{E \vdash a = b}\ \textit{Substitutivity}}{E \vdash f(a) = f(b)}\ \textit{Congruence} \qquad \dfrac{}{E \vdash f(b) = g(b)}\ \textit{Substitutivity}}{E \vdash f(a) = g(b)}\ \textit{Transitivity}$$

Example 6.2 (자연수 덧셈 예시)

`NAT-ADD ⊢ s(s(0)) + s(0) = s(0) + s(s(0))`

즉, "2 + 1 = 1 + 2"가 논리적으로 성립함을

13단계 deduction으로 보여줌 (계산처럼 차근차근).

1.	NAT-ADD $\vdash$ s(s(0)) + s(0) = s(s(0) + s(0))	(E_1; s(M) + N = s(M + N))
2.	NAT-ADD $\vdash$ s(0) + s(0) = s(0 + s(0))	(E_1; s(M) + N = s(M + N))
3.	NAT-ADD $\vdash$ 0 + s(0) = s(0)	(E_1; equation 0 + M = M)
4.	NAT-ADD $\vdash$ s(0 + s(0)) = s(s(0))	(E_5; from 3)
5.	NAT-ADD $\vdash$ s(0) + s(0) = s(s(0))	(E_4; from 2, 4)
6.	NAT-ADD $\vdash$ s(s(0) + s(0)) = s(s(s(0)))	(E_5; from 5)
7.	NAT-ADD $\vdash$ s(s(0)) + s(0) = s(s(s(0)))	(E_4; from 1, 6)
8.	NAT-ADD $\vdash$ s(0) + s(s(0)) = s(0 + s(s(0)))	(E_1; s(M) + N = s(M + N))
9.	NAT-ADD $\vdash$ 0 + s(s(0)) = s(s(0))	(E_1; equation 0 + M = M)
10.	NAT-ADD $\vdash$ s(0 + s(s(0))) = s(s(s(0)))	(E_5; from 9)
11.	NAT-ADD $\vdash$ s(0) + s(s(0)) = s(s(s(0)))	(E_4; from 8, 10)
12.	NAT-ADD $\vdash$ s(s(s(0))) = s(0) + s(s(0))	(E_3; from 11)
13.	NAT-ADD $\vdash$ s(s(0)) + s(0) = s(0) + s(s(0))	(E_4; from 7, 12)

- 핵심 : 등식이 성립함을 보이려면 규칙 적용의 유한한 수열(deduction chain)을 주면 됨.
- 하지만 반대로, 등식이 성립하지 않음을 증명하는 것은 불가능함.

  왜냐하면 "이제 더 이상 안 되네"가 참인지 아닌지 기계적으로 확인 불가하기 때문.

Theorem 6.1 — 결정 불가능성(Undecidability)

"E ⊢ t = u"가 성립하는지 판정할 수 있는 알고리즘은 존재하지 않는다.

즉, 어떤 식이 증명 가능한지를 기계적으로 알아내는 것은 불가능.

(수학적으로는 참이 존재하지만, 그것을 **자동으로 판정**할 수는 없음.)

이건 **튜링의 정지 문제(Halting Problem)** 와 본질적으로 동일한 성격이다.

Theorem 6.2 — Equational Logic = Rewriting System

E ⊢ t = u ⇔ t ↔* u (서로 rewrite로 변환 가능하면 논리적으로도 같음)

이 정리는 "rewrite 시스템"과 "논리 증명 시스템"이 같은 의미임을 보임.

즉, **논리적 증명(E ⊢ t = u)** ≡ **rewrite 과정으로 t를 u로 바꾸는 것**

Theorem 6.3 — 여전히 결정 불가능

t ↔* u가 성립하는지도 일반적으로 결정 불가능이다.

직관적으로, rewrite로 서로 변환 가능한지 확인하려면 무한히 시도해야 할 수도 있기 때문.

결국, rewrite 시스템의 **합류성(confluence)**도 결정 불가능하다.

Theorem 6.4~6.5 — 결정 가능한 특별한 경우

단, 만약 시스템이

- **terminating (모든 rewriting이 유한 단계 안에 끝남)**
- **confluent (모든 경로가 같은 결과로 합쳐짐)**이면 다음이 성립한다:

  t ↔* u ⇔ t! = u!

$E \vdash t = u$ ⇔ $t! = u!$

즉, **정상형(normal form)** $t!$, $u!$ 만 비교하면 됨.

"rewrite 다 해보고 마지막 결과가 같으면 논리적으로 같다!"

6.1.1 Kunth-Bendix Completion

배경 — 왜 필요한가?

- 등식 논리에서 $E \vdash t = u$ (즉, "t와 u가 논리적으로 같다")는 **termination**(모든 rewrite가 멈춤) + **confluence**(모든 경로가 합쳐짐)이라는 조건 아래에서는 쉽게 판정 가능하다.
- 하지만, 대부분의 명세(specification)는 **terminating도 아니고 confluent도 아니다.**
- 그래서 **Knuth–Bendix Completion**은 비합류적(non-confluent) 또는 비종료적(non-terminating) 시스템을 논리적으로 동등한 **terminating + confluent** 시스템으로 변환하려는 과정이다.

핵심 아이디어

- 합류적이지 않은 이유

  시스템이 합류적이지 않다는 것은, 서로 다른 정상형을 만드는 **critical pair (t, u)**가 존재한다는 뜻이다.

  즉, t →* t'   그리고   u →* u'   하지만   t' ≠ u'

- 해결 방법

  이 두 정상형을 **같게 만드는 새로운 등식**을 추가한다. → t' = u'   (또는 반대 방향)

  이렇게 해서 충돌(conflict)을 없애 나간다.

- 반복

  모든 critical pair가 해결되고 새 등식들이 **감소 순서(>)** 를 만족하며 더 이상 합류되지 않는 쌍이 없을 때, completion이 "성공적으로 종료된다".

Example 6.4 — 그룹 공리

그룹의 기본 등식:

```
G = { e ∘ x = x,   i(x) ∘ x = e,   (x ∘ y) ∘ z = x ∘ (y ∘ z) }
```

→ **합류적이지 않음**

Knuth–Bendix Completion을 적용하면 다음과 같은 **동등한** 명세로 변환됨:

```
{ e ∘ x = x,        i(x) ∘ x = e,        (x ∘ y) ∘ z = x ∘ (y ∘ z),
  i(x) ∘ (x ∘ y) = y,  x ∘ e = x,          i(e) = e,
  i(i(x)) = x,        x ∘ i(x) = e,        x ∘ (i(x) ∘ y) = y,
  i(x ∘ y) = i(y) ∘ i(x) }
```

이제 이 시스템은 **terminating + confluent** 하므로, $t = u$ 의 성립 여부를 **정상형 비교(**$t! = u!$**)** 로 결정할 수 있다.

즉, 그룹 이론에서의 등식 판정은 결정 가능(decidable) 하다.

하지만 항상 성공하는 건 아니다

Knuth–Bendix Completion은 모든 경우에 성공하지 않는다.

- 시스템이 너무 복잡하면 **completion이 멈추지 않을 수 있음.**

  (새로운 식이 계속 생기며 무한 반복)

- 어떤 등식은 방향을 정하기 어렵거나 **감소 순서(termination ordering)** 를 만족하지 못할 수 있음.
- 따라서, **completion이 종료되지 않거나, 여전히 결정 불가능** 할 수도 있다.

6.2 Inductive Theorems

핵심 개념

- **Equational Logic** (등식 논리)은 어떤 등식이 **모든 모델(all models)**에서 참인지 판단한다.

  → 즉, E를 만족하는 모든 구조에서 `E ⊢ t = u` 가 참이면 "논리적으로 참"이다.

- 하지만 우리가 진짜 관심 있는 건, 현실적으로 의미 있는 "의도된 모델(intended model)" 안에서만 참인 성질이다.

  예: 자연수 덧셈(`NAT-ADD`) 모델에서만 `m + n = n + m` (교환법칙)이 성립해야 함.

문제: 일반 등식 논리의 한계

예를 들어, NAT-ADD ⊢ M + N = N + M은 **거짓**이다.

왜냐하면 NAT-ADD를 만족하는 이상한 모델(예: `a + 0 ≠ 0 + a`)도 존재하기 때문.

즉, "모든 구조"에서는 교환법칙이 깨질 수 있다.

그러나 우리가 원한 건 "자연수 안에서만" 참인 것. 즉, `0` , `s(0)` , `s(s(0))` , ... 과 같은 **constructor ground term**(실제 데이터 값)에 대해서만 참이길 바란다.

귀납적 정리(Inductive Theorem)

따라서, 모든 **constructor ground term** `m` , `n` 에 대해 다음이 성립하면 (여기서 모든 constructor ground term은 의도된 모델을 말함)

NAT-ADD ⊢ m + n = n + m

이걸 **inductive theorem (귀납적 정리)** 라고 한다.

즉, E ⊢ind t = u ⇔ 모든 constructor ground term 치환 σ에 대해 `E ⊢ tσ = uσ` 가 참

이게 바로 "의도된 모델(intended model)"에서만 참인 등식이다.

예제 (Example 6.5)

- 일반적인 등식 논리에서는 BOOLEAN ⊢ Y implies X = (not Y) or X가 성립하지 않음.

  (모든 가능한 구조에서 참은 아님)

- 그러나 귀납적 정리에서는 BOOLEAN ⊢ind Y implies X = (not Y) or X는 성립한다.

  (의도된 Boolean 모델 내에서는 참)

"좋은 증명 시스템(optimal proof system)"의 조건

귀납적 정리를 자동으로 증명할 수 있다면 정말 좋겠지만,

그걸 위해서는 다음 세 조건을 모두 만족해야 함.

조건	설명
Sound	증명된 것은 실제로 참이어야 함 (거짓을 증명할 수 없음)

조건	설명
Complete	참인 것은 모두 증명할 수 있어야 함
Algorithmically checkable	어떤 증명이 주어졌을 때, 그게 올바른지 기계적으로 검사 가능해야 함

등식 논리는 이 세 조건을 모두 만족하지만,

귀납적 정리(Inductive Theorem) 에 대해서는 이런 시스템이 존재하지 않는다.

이유 — 힐베르트의 10번째 문제와 결정 불가능성

그 이유는 **결정 불가능성(undecidability)** 때문이다.

즉, “모든 귀납적 정리를 증명하거나 거짓임을 판정하는 알고리즘은 존재하지 않는다.”

이건 Hilbert의 10번째 문제(“디오판틴 방정식이 해를 갖는지 결정 가능한가?”)의 부정적 결과로 증명됨.

1970년, **Yuri Matiyasevich**(당시 22세)가 Davis–Putnam–Robinson의 연구를 완성함.

6.2.1 Proving Inductive Theorems for Nat

Inductive Theorem이란?

- 우리가 다루는 **“의도된 모델(intended model)”** (예: 자연수, 리스트 등)에 대해, 어떤 등식이 모든 모델에서 아니라 **그 모델에서만 참**임을 보이고 싶을 때 사용.
- 즉, E ⊢_ind t = u라는 건 “모든 constructor ground term에 대해 t = u가 참”이라는 뜻.

예제 1: 단순한 귀납 정리 — (x + 0 = x)

- **목표**

  NAT_ADD ⊢_ind x + 0 = x

- **아이디어**
  - 귀납변수: x
  - 귀납 기초(Base): 0 + 0 = 0
  - 귀납 단계(Induction Step): s(t) + 0 = s(t)

- **증명 흐름**

  s(t) + 0 → s(t + 0) ↔(Ind.hyp.) s(t)   (귀납가정: t + 0 = t)

- **결론**

  NAT_ADD ⊢ t + 0 = t

  따라서 귀납적으로 x + 0 = x 성립

일반적인 귀납 증명 틀

귀납으로 어떤 성질 (P(t))를 보일 때는:

1. **Base case**: (P(0))이 참임을 보인다.
2. **Induction step**: (P(t))가 참이라고 가정하고 (P(s(t)))를 증명한다.

때로는 더 강한 귀납 가정(예: 깊이가 더 작은 모든 항에 대해 성립)을 쓸 수도 있다.

예제 2: 덧셈의 결합법칙 (Associativity)

- **목표**

  NAT_ADD ⊢_ind (x + y) + z = x + (y + z)

- **귀납 변수**

  t_1

- **증명 단계**
  - **기초 단계**

    ```
    (0 + t2) + t3 → t2 + t3
    0 + (t2 + t3) → t2 + t3
    ```

    → 양쪽 동일 normal form 도달 → 참

  - **귀납 단계**

    귀납 가정 : (t + t2) + t3 = t + (t2 + t3)

    증명 목표 : (s(t) + t2) + t3 = s(t) + (t2 + t3)

    이건 `s(M) + N = s(M + N)` 규칙으로 바로 도달 가능.

  - **결론**

    결합법칙은 귀납적으로 성립

예제 3: 덧셈의 교환법칙 (Commutativity)

- **목표**

  NAT_ADD ⊢_ind t_1 + t_2 = t_2 + t_1

- **귀납 변수**

  t_1

- **문제점**

  단순 rewrite로는 $(s(t) + t_2 \rightarrow s(t + t_2))$까지만 가고, 우변 $(t_2 + s(t))$에는 **절대 도달할 수 없음**

  → 역방향 변환을 도와줄 **보조정리(lemma)** 가 필요함.

- **필요한 보조정리들 (Lemmas)**

이름	내용	역할
Lemma 1	t + 0 = t	기본 귀납 정리 (이미 증명됨)
Lemma 2	s(t_1 + t_2) = t_1 + s(t_2)	rewrite 역방향을 연결하는 다리 역할

- 교환법칙 증명 흐름

  Base case : 0 + t_2 = t_2 + 0

  좌변 : 0 + t_2 → t_2

  우변 : t_2 + 0 → t_2 (Lemma 1)

  → 같음

  Induction step:

  귀납가정 : t + t_2 = t_2 + t

목표 : s(t) + t_2 = t_2 + s(t)

rewrite 순서 :

```
s(t) + t₂ → s(t + t₂)
→ s(t₂ + t)        (귀납가정)
→ t₂ + s(t)        (Lemma 2)
```

→ 증명 완료

6.2.2 Inductive Theorems for Other Data Types

핵심 개념

- `NAT-ADD`에서 했던 귀납 증명(Inductive Proof)은 **임의의 데이터 타입**에도 일반화 가능하다.
- 어떤 성질 P(t)이 **constructor ground term** t에 대해 참임을 보이려면:
    - **기초 단계(Base case)**: 모든 constructor **상수** c에 대해 P(c)가 성립함을 보인다.
    - **귀납 단계(Induction step)**:
        - 깊이가 n+1인 term f(t_1, ..., t_n)에 대해 P(f(t_1, ..., t_n))을 증명한다.
        - 이때 깊이가 더 작은 t_i들에 대해 P(t_i)를 귀납 가정으로 사용할 수 있다.

예제 6.8 — 임의 타입 M의 귀납 정리

데이터 타입 정의

```
fmod M is
  sorts s s' .
  ops a b : → s [ctor] .
  ops f g : s s' → s [ctor] .
  op k : s s' s → s [ctor] .
  ops l p : s → s .
  ops c d : → s' [ctor] .
  op h : s' s s' → s' [ctor] .
  ops d : s → s' .
  ... variables and equations ...
endfm
```

목표

성질 Q(t)이 sort s의 모든 constructor ground term t에 대해 성립함을 보이고자 한다.

증명 구조

1. **Base case**

   Q(a)와 Q(b)를 증명한다. (상수 constructor의 경우)

2. **Induction step**

    - Q(f(t,t')), Q(g(t,t')) :

      임의의 t, t'에 대해 성립함을 보이고, Q(t)를 귀납 가정으로 사용한다.

- Q(k(t_1,t_2,t_3)) :

  Q(t_1)과 Q(t_3)를 모두 가정하고 증명한다.

→ 즉, 각 constructor의 인자 구조에 따라 필요한 귀납 가정을 모두 세워야 함을 보여주는 예시.

예제 6.9 — Binary Tree에 대한 귀납 증명

목표

이진 트리 `t`에 대해 다음이 항상 성립함을 증명 : BINTREE-NAT1 ⊢ size(reverse(t)) = size(t)

즉, 트리를 뒤집어도 노드의 개수는 변하지 않는다.

트리 정의

```
fmod BINTREE-NAT1 is
  sort BinTree .
  op empty : → BinTree [ctor] .
  op bintree : BinTree Nat BinTree → BinTree [ctor] .
  ops size weight : BinTree → Nat .
  op reverse : BinTree → BinTree .

  eq size(empty) = 0 .
  eq size(bintree(BT, N, BT')) = s(0) + (size(BT) + size(BT')) .
  eq reverse(empty) = empty .
  eq reverse(bintree(BT, N, BT')) = bintree(reverse(BT'), N, reverse(BT)) .
endfm
```

증명 단계

1. **Base case**

   size(reverse(empty)) = size(empty)

   → (0 = 0), 자명함

2. **Induction step**

   size(reverse(bintree(t_1, n, t_2))) = size(bintree(t_1, n, t_2))

   - 귀납 가정:
     - size(reverse(t_1) = size(t_1)
     - size(reverse(t_2)) = size(t_2)

Maude 코드

```
fmod PROVE-BINTREE is including BINTREE-NAT1 .
  ops t1 t2 : → BinTree .
  op n : → Nat .
  eq size(reverse(t1)) = size(t1) .  --- Ind. Hyp.
  eq size(reverse(t2)) = size(t2) .  --- Ind. Hyp.
endfm
```

```
red size(reverse(empty)) == size(empty) .
red size(reverse(bintree(t1, n, t2))) == size(bintree(t1, n, t2)) .
```

결과 해석

- 첫 번째 줄: `true` 반환 → 기초 단계 확인 완료.
- 두 번째 줄:

  Maude는 다음을 계산 : (size(t_2) + size(t_1)) == (size(t_1) + size(t_2))

  → 덧셈의 **교환법칙**이 이미 증명되어 있으므로 참임.