4. Termination

- **종결성(Termination)의 정의**
  - 어떤 등식 사양 E가 종결적이라는 것은 **모든 항 t_0에 대해 무한한 유도 과정이 존재하지 않음**을 의미한다.
  - 즉, 모든 유도 과정은 반드시 유한해야 한다.
- **간단한 예시**
  - {f(x)=g(x)}, {f(g(x))=h(x)}, {a=b,b=c}→ 종결적.
  - {f(x,y)=f(y,x)}, {f(x)=f(g(x))}, {a=b,b=c,b=a} → 종결적이지 않음(무한 rewrite 발생).
  - 일부 경우는 **weakly terminating** (모든 항이 정상형을 갖지만, 무한 유도도 가능).
- **종결성 판별 문제**
  - "`bool terminates(E)`" 같은 판별 알고리즘이 있으면 좋겠지만, **튜링 기계와 동등**하므로 일반적으로 **결정 불가능(undecidable)**.
  - 따라서 어떤 사양이 종결적인지 **항상 자동으로 판별할 수 있는 알고리즘은 존재하지 않는다.**
- **실제로 할 수 있는 것**
  - **비종결성(non-terminating)** 증명 → 무한 유도 과정 하나만 찾으면 된다.
  - **종결성(terminating)** 증명 → 모든 초기 항에 대해 유한하게 끝남을 보이면 된다.
- **후속 내용 (4.2~4.4에서 다룸)**
  - **4.2**: 비종결성을 증명하는 방법.
  - **4.3**: 종결성을 증명하는 방법(각 term에 "weight"를 부여하고 유도가 weight 감소를 보이는지 확인).
  - **4.4**: 강력한 단순화 순서(simplification orders) 이론 및 자동화 방법.

4.1 Undecidability of Termination

1. 종결성(termination)의 결정 불가능성

- Church–Turing 명제에 따르면, 모든 계산 가능한 함수는 튜링 기계로 표현 가능.
- **종결성 판정 문제**: 어떤 튜링 기계가 모든 입력에서 멈추는지 여부는 **결정 불가능(undecidable)**.
- 따라서 등식 사양 E의 종결성 여부도 일반적으로는 판별할 수 없다.
- 정리 4.1: "*어떤 사양이 종결적인지 여부는 undecidable 이다.*"

---

2. 튜링 기계 정의 (recap)

- M=(Q,S,Δ)
  - Q: 유한 상태 집합
  - S: 알파벳 (공백 포함)
  - Δ: 전이 관계
- 동작: 헤드가 기호를 읽고, 새 기호를 쓰고, 상태 변경 후 좌/우로 이동.

---

3. Maude에서의 표현

- 튜링 기계는 리스트/테이프를 term rewriting system으로 모델링 가능.
- 상태, 기호, 구분자를 term으로 정의하고, 전이 규칙을 Maude 식으로 표현.
- 예:
  - (q, s, q', s', left/right)

---

4. 두 가지 문제점

1. **Order-sorted vs Unsorted**
   - 실제 튜링 기계는 order-sorted 사양으로 표현되지만, 우리는 unsorted 사양에서 종결성을 다뤄야 함.
   - 변환(sorted → unsorted) 방법은 간단 (리스트를 unary 함수 형태로 변환).
2. **Junk terms 문제**
   - e(M)이 종결적이려면 모든 초기 term t_0에 대해 멈춰야 함.
   - 여기에는 실제로 존재하지 않는 잘못된 term(= junk term)도 포함.
   - 문제: M은 종결적이라도, junk term에서 e(M)이 무한 루프를 돌면 e(M)은 비종결적.

---

5. 예시: M_ab

- M_ab: 'a', 'b' 확인 및 보정하는 간단한 튜링 기계.
- 정상 입력에서는 항상 멈춤 (종결적).
- 그러나 "junk term"으로 여러 헤드를 동시에 둔 경우(예: [q_init a q_init b]),

  → 두 헤드가 서로 충돌하여 무한 루프 발생 → e(M_ab) 비종결적.

---

6. 해결책

- 여러 "인스턴스"가 **상호작용(interact)** 하지 못하게 해야 함.
- Baader & Nipkow의 아이디어: 각 기호를 헤드에 따라 구분 표시 (s←,s→).
- 전이는 자기 헤드를 가리키는 기호만 다룸.
- 따라서 한 헤드가 다른 헤드의 기호를 덮어쓰지 않음 → junk term에서도 무한 루프 방지.
- 이 수정으로, M과 e~(M)은 종결성이 **정확히 일치**.

4.2 Nontermination

- **Looping 정의**:

  사양 (E)가 looping이라 함은,

$$t \Rightarrow_E^+ u \quad \text{이고 } t \subseteq u$$

  (즉, 유도 후 결과 (u) 안에 원래 (t)가 다시 부분항으로 들어감).

  → 이런 경우 무한 반복 가능 → **비종결적(nonterminating)**.

예제

- $\{f(x) = f(f(x))\}$:
  $f(x) \to f(f(x)) \to f(f(f(x))) \to \cdots$ → looping → nonterminating.
- $\{f(x, y) = f(y, x)\}$:
  $f(x, y) \leftrightarrow f(y, x)$ 무한 반복 → looping → nonterminating.
- $\{f(x) = g(x, y)\}$:
  오른쪽에 새로운 변수 등장 → $f(x) \to g(x, f(x)) \to g(x, g(x, f(x))) \to \cdots$
  → nonterminating (looping 포함).

중요한 규칙

- **등식 오른쪽에 새로운 변수를 도입하면 안 됨.** (무한 유도의 원인이 됨)

Looping이 아닌데도 비종결인 경우

$\{f(x) = f(g(x))\}$:
$f(x) \to f(g(x)) \to f(g(g(x))) \to \cdots$
→ **nonterminating but not looping**.

정리

- **Looping ⇒ Nonterminating** (항상).
- **Nonterminating ≠ 항상 Looping** (loop 없이도 무한 유도 가능).
- 오른쪽에 새로운 변수는 절대 금지.

4.3 Proving Termination Using “Weight Functions”

1. 아이디어

- **Termination 증명 방법**: 각 항(term)에 “가중치(weight)”를 부여.
- Rewrite가 일어날 때마다 **가중치가 줄어든다**면 무한 반복이 불가능 → termination 보장.

---

2. 기본 정의

- **Weight Function**:
  $weight : T_\Sigma \to \mathbb{N}$
  모든 rewrite $t \Rightarrow u$에 대해

$$weight(t) > weight(u)$$

  → 사양은 termination.
- **Monotonicity**:
  Context 안에 들어가도 weight 감소가 유지되어야 함.
  즉, $w(t) > w(u) \implies w(f(\dots, t, \dots)) > w(f(\dots, u, \dots))$.
- **Proposition 4.2**:
  모든 등식 $l = r$과 ground substitution σ에 대해
  $weight(l\sigma) > weight(r\sigma)$가 성립하면 termination.

---

3. 예제

- 예제 **4.7**: $\{f(x) = g(x)\}$
  - weight = 항 안의 f 개수
  - 단조적이고, $f(x) \to g(x)$ 시 weight 감소 → termination 증명.
- 예제 **4.8**: $\{f(x) = g(x), g(b) = f(a)\}$
  - $weight(a) = 1, weight(b) = 88, weight(f(t)) = 4 + weight(t), weight(g(t)) = weight(t)$.
  - 각 규칙에서 좌변 weight > 우변 weight → termination.
- 예제 **4.9**: $\{f(g(x)) = g(f(x))\}$
  - $weight(a) = 2, weight(f(t)) = (weight(t))^3, weight(g(t)) = 2 \cdot weight(t)$.
  - 각 rewrite에서 weight 감소 → termination.
- 예제 **4.10**: $\{f(f(x)) = f(g(f(x)))\}$
  - 단조적 weight 정의는 어려움.
  - 대신 "adjacent f 쌍 개수"라는 **비단조적 weight** 사용 가능.

---

4. 확장 (Well-founded Order)

- 자연수 외에도 **well-founded strict partial order**를 weight로 쓸 수 있음.
- 조건: 무한 감소열이 없어야 함.
- 예:
  - 자연수의 ">"는 well-founded ($\mathbb{N}$에서는 ok, $\mathbb{Z}$에서는 아님).
  - 사전식 순서(lex order): 각 도메인이 well-founded하면 튜플/리스트도 well-founded.
  - Multiset order: 공통 원소 제거 후 더 큰 원소가 남은 쪽이 큰 것.

4.4 Simplification Orders

배경

- Weight function으로 종료성(termination)을 증명하려면 "적절한 weight 함수"를 찾는 아이디어가 필요 → 자동화에는 적합하지 않음.
- 따라서 **simplification order (단순화 순서)** 이론을 도입 (Dershowitz에 의해 제안됨).
- 이 접근은 더 강력하며 자동화가 가능함.

Embedding (포함 관계)

- **아이디어**: term `t`가 term `u`를 *포함(embed)* 한다는 것은, `t` 안에 `u`가 부분항(subterm)으로 들어있음을 의미.
- 즉, `t`에서 일부 함수 심볼들을 지우면 `u`를 얻을 수 있는 경우.

Formal Definition (Def 4.5)

- 표기: `t ▹ u` ("t embeds u").
- 의미: `t`에서 특정 규칙을 따라 `u`로 변환할 수 있으면 `t ▹ u`.
- 변환 규칙은 다음과 같은 **EMB** 집합으로 정의

$$EMB = \{f(x_1, ..., x_m) = x_i \mid 1 \le i \le m\} \cup \{g(x_1, ..., x_n) = x_i \mid 1 \le i \le n\} \cup ...$$

  즉, 모든 non-constant 함수에 대해, 인자를 하나 선택해서 그걸로 대체할 수 있다는 뜻.

  → `f(x1,...,xm)`에서 `xi`만 남기고 나머지는 날리는 규칙.
- `t ▹_ u` 는 `t`가 여러 번의 EMB 변환(`t ⇝* u`)으로 `u`에 도달 가능함을 의미.

Example 4.12

1. `f(g(f(a))) ▹ f(f(a))`
   - 규칙 `g(x1) = x1`을 적용하면 성립.
2. `f(a, g(h(b, f(c,d)), e)) ▹ f(a, h(b,d))`
   - 중간 인자 정리해서 `h(b,d)` 부분만 남길 수 있음.
3. 그러나 `f(a,g(h(b,f(c,d)),e)) ▹ f(a,h(b,e))` 는 안됨.

즉, "embedding"은 함수 껍질을 벗겨내며 안쪽 인자를 꺼내올 수 있는 규칙으로 생각하면 됨

Theorem 4.2 (Kruskal's Tree Theorem)

1. 내용

- Σ가 **유한한 함수 기호 집합**이라면,
- 임의의 무한 항 수열 (t_1, t_2, ..., t_j, ..., t_k, ...) 안에는 반드시 **(j < k)이면서 (t_k `▹_` t_j)**인 두 항이 존재한다.

즉, 무한히 항을 만들면 언젠가는 "뒤 항이 앞 항을 포함(embed)"하는 순간이 생긴다.

2. 의미

- 만약 어떤 사양이 무한 실행(=non-termination)을 가지려면, 그 실행 도중 반드시 **self-embedding** (뒤 항이 앞 항을 포함) 패턴이 발생해야 한다.
  - 왜 : 무한히 달리려면 자기 꼬리를 물고 돌아와야 하는데, 그 꼬리무는 패턴(self-embedding)을 막아버리면 달리기를 영원히 할 수 없다. ⇒ 결국 멈춘다.
- 따라서 **self-embedding이 발생하지 않는 사양은 termination 한다.**

3. 증명 아이디어

1. 무한 유도열이 있다고 가정하면:

   t1 > t2 > ⋯ > t_j > ⋯ > t_k > ⋯ (>는 strict partial order)

2. Kruskal 정리에 따라 t_k ▸_ t_j인 j, k가 반드시 존재.
3. 그런데 두 가지 경우가 모두 모순:
   - **Case 1**: t_k = t_j → t_j > t_k 라 모순.
   - **Case 2**:

     $t_k \triangleright t_j$ → 이미 $t_j \succ t_k$가 있는데, 동시에 $t_k \succ t_j$이면 추이성으로 $t_j \succ t_j$가 되어 모순.

4. 따라서 무한 유도열 자체가 불가능. → 즉, termination.

4. 결론

- **Kruskal's Tree Theorem**은 "유한한 함수 기호 집합에서 무한히 새로운 항만 만들 수는 없다. 결국 포함관계가 생긴다."는 것을 보장한다.
- 이 사실을 이용하면, **termination 증명을 자동화**할 수 있다.
- 핵심은: **self-embedding 없음 ⇒ termination 보장.**

5. 한 줄 요약

"Kruskal의 tree 정리에 따르면, 무한 실행에는 반드시 self-embedding이 필요하다. 따라서 self-embedding이 없는 사양은 무한 실행이 불가능하고, 반드시 termination 한다."

Termination을 자동으로 증명하는 방법 = "단순화 순서 (simplification order)"라는 특수한 순서를 정의해서 쓰자.

1. Simplification order가 뭐냐?

- 어떤 항 f(t_1, ..., t_n) 있다고 하자.
- **subterm property**: 전체 항은 언제나 자기 부분항보다 크다.

  f(t1, ..., tn) > ti
- 이 순서를 **simplification order**라고 부른다.

2. 왜 중요하냐?

- 우리가 termination을 증명하려면, rewrite t ⇒ u 할 때마다 "항이 점점 작아진다"는 걸 보이면 됨.
- 그걸 수학적으로 보장하는 게 이 **>(simplification order)**임.

3. 명제 4.3

- 만약 (t ▹ u) (embedding 관계: t가 u를 포함)라면, 단순화 순서에서는 반드시 t > u가 성립한다.
- 즉, "포함 관계가 있으면 그건 strict order에서 더 큰 거다"라는 사실.

4. 정리 4.3

- 함수 기호나 방정식이 유한하다면,
- 모든 rewrite 규칙 (l = r)와 치환 σ에 대해 (lσ > rσ)가 성립한다면,
- 그 사양은 termination 한다.
- 즉, **모든 rewrite가 순서를 따라 줄어든다면 → 무한 실행 불가능 → termination.**

5. 한계

- Simplification order는 “self-embedding이 없음을” 보장하는데 특화돼 있음.
- 하지만 (f(f(x)) = f(g(f(x)))) 같은 경우처럼 **self-embedding도 있고 termination도 되는 예외적 시스템**은 증명 못함.

6. 요약

- **Simplification order** = “항이 부분항보다 크다”라는 순서.
- 이 순서를 따라 rewrite가 항상 줄어든다면 termination.
- Kruskal 정리 덕분에 이게 잘 작동한다.
- 하지만 모든 경우(특히 self-embedding + terminating 케이스)는 다 증명 못한다.

4.4.1 Lexicographic Path Order (LPO)

개념

- **Lexicographic Path Order (LPO)**: 강력한 **simplification order** 중 하나로, 자동으로 적용 가능.
- 필요조건: 함수 심볼들에 대해 **precedence(우선순위)**라는 strict partial order가 정의되어야 함.

정의 (Definition 4.7)

LPO는 다음 조건들을 만족하는 가장 작은 관계:

1. **(lpo-1)**:
   만약 어떤 부분항 $t_i \succ_{lpo} u$ 이거나 $t_i = u$ 라면,
   $$f(\ldots, t_i, \ldots) \succ_{lpo} u$$
2. **(lpo-2)**:
   만약 $f \succ g$ 이고, 모든 $i \leq m$ 에 대해 $f(t_1, \ldots, t_n) \succ_{lpo} u_i$ 이라면,
   $$f(t_1, \ldots, t_n) \succ_{lpo} g(u_1, \ldots, u_m)$$
3. **(lpo-3)**:
   $(t_1, \ldots, t_n) \succ_{lpo}^{lex} (u_1, \ldots, u_n)$ 이고,
   각 $2 \leq i \leq n$ 에 대해 $f(t_1, \ldots, t_n) \succ_{lpo} u_i$ 라면,
   $$f(t_1, \ldots, t_n) \succ_{lpo} f(u_1, \ldots, u_n)$$

Proposition 4.4

>lpo는 **모든 precedence**에 대해 simplification order가 된다.

확장

- LPO는 변수가 포함된 항에도 확장 가능.
- 변수는 precedence에서 **비교 불가능한 상수**로 처리.
- 이 경우, l>lpo r ⇒ l_σ >lpo r_σ (모든 치환 σ에 대해).

활용

- termination을 보이려면, 함수 심볼들에 precedence를 정의해서 모든 등식 l = r 에 대해 l >lpo r 임을 보이면 됨.
- 예: 곱셈은 덧셈으로 정의되고, 거듭제곱은 곱셈으로 정의되므로, precedence를 **>*>+ 로 두면 termination 증명이 가능.

Example 4.13

다음 시스템의 termination 증명:

$$\{0 + x = x, \quad 0 * x = 0, \quad x * *0 = s(0), \quad s(x) + y = s(x + y), \quad s(x) * y = y + (x * y), \quad x * *s(y) = x * (x * *y)\}$$

- 각각의 방정식이 >lpo-decreasing 임을 보임으로써 종료를 증명.

특징

- **자동화 가능**: 유한한 함수 심볼 집합에 대해서는 가능한 precedence도 유한 개.
- 각 등식이 >lpo-decreasing인지 확인하는 과정 자체가 termination 증명.
- 따라서 프로그램이 모든 precedence에 대해 LPO termination 여부를 자동으로 검사할 수 있음.

4.4.2 The Multiset Path Order and Other Variations of lpo

LPO (Lexicographic Path Order)

- 사전식 비교
- `(t1, t2, t3)` 와 `(u1, u2, u3)` 를 비교할 때:
  - 먼저 **첫 번째 항** `t1` vs `u1` 비교
  - 같으면 두 번째 항 `t2` vs `u2` 비교
  - 이런 식으로 **순서대로 차례차례 비교**.

→ 즉, **순서가 중요**함.

MPO (Multiset Path Order)

- 다중집합 비교
- `(t1, t2, t3)` 와 `(u1, u2, u3)` 를 비교할 때:
  - 순서를 무시하고, 그냥 집합처럼 {t1, t2, t3} vs {u1, u2, u3} 로 비교.
- 그래서 `t1` 은 `u3` 랑, `t2` 는 `u1` 이랑 짝지어 비교할 수도 있음.

→ 즉, **순서는 상관없고 전체 집합적으로 비교**함.

왜 두 가지가 필요한가?

- LPO만으로는 증명 안 되는 종료성이 있고,
- MPO만으로는 증명 안 되는 종료성이 있어요.
- 그래서 서로 보완적임.
- 예:
  - `{ f(a,b) = f(b,a) }`
    → 항 순서가 바뀌는 거니까 **LPO로는 종료 증명 가능**, MPO는 못함.
  - `{ g(x,a) = g(b,x) }`
    → 집합적으로 위치가 바뀌는 거니까 **MPO로는 종료 증명 가능**, LPO는 못함.

요약

- **LPO**: 순서를 지키며 비교 (사전식).
- **MPO**: 순서 무시하고 집합으로 비교 (다중집합).
- 서로 다른 케이스에서 종료성을 증명할 수 있음 → 그래서 같이 쓰거나 확장해서 씀.

4.4.3 Comparing Weight Functions and Simplification Orders

1. 차이점

- **Weight Function (가중치 함수)**
  - 각 항(term)에 숫자(가중치)를 직접 정의해서 "항이 점점 작아진다"는 걸 증명.
  - 매번 새롭게 설계해야 하므로 **손으로 정의해야 하고 복잡할 수 있음**.
  - 하지만 어떤 경우엔 매우 강력 (예: self-embedding 시스템 종료 증명 가능).
- **Simplification Orders (단순화 순서: LPO/MPO)**
  - 미리 정의된 순서 규칙(LPO, MPO 등)을 자동으로 적용.
  - **자동화 가능** → 증명하기 쉬움.
  - 하지만 약점: self-embedding 같은 특정 시스템에 대해서는 종료성을 보장하지 못함.
    - LPO/MPO는 self-embedding을 "막는" 도구라서, 실제로 self-embedding 구조가 있으면 termination 증명 불가.

2. 강점

- **Path Orders (LPO, MPO)**
  - 꽤 강력해서 Ackermann 함수 같은 것도 종료 증명 가능.
  - 예: `{ f(s(y), x, z) = f(y, x+y+z, z+z) }` 같은 시스템은 path order로는 증명됨.
  - 하지만 polynomial weight function으로는 불가능.
- **Weight Functions**
  - `{ f(g(h(x))) = f(f(x)), f(g(h(x))) = g(g(x)), f(g(h(x))) = h(h(x)) }` 같은 시스템은 LPO/MPO로는 안 되지만, weight function으로는 가능.

3. 정리

- **LPO/MPO**: 자동화 쉽고 강력하지만, self-embedding 시스템에선 약함.
- **Weight Function**: 직접 설계해야 하지만, self-embedding 포함 더 일반적인 케이스 처리 가능.
- 결국 두 가지는 상호 보완 관계.