3. Operational Semantics of Equational Specifications

2장에서는 *Maude*에서 등식 명세(equational specification)를 **쓰는 방법**을 보여줬지만, 그 **정확한 의미 (semantics)** 는 설명하지 않았음.

이 **3장**에서는 등식 명세가 어떤 의미를 가지는지, 즉 **연산적 의미(operational semantics)** 를 다룸.

Operational semantics = "계산적 의미"

즉, 명세(specification)를 **실행했을 때 어떻게 동작하는지** 설명하는 것.

여기서는 이렇게 정의함:

- 어떤 ground term `t` 가 있으면, 명세 안의 어떤 등식을 이용해서 한 단계에 term `t'` 로 **줄어든다(reduces)**.

단순화를 위해 이 장에서는:

1. **unsorted (or one-sorted)** = 정렬이 없음 → 명세에 **하나의 sort만 존재**
2. **unconditional equations** = 조건 없는 등식만 다룸
3. **함수 기호(function symbols)에 속성 없음** → 예: `assoc` (결합법칙), `comm` (교환법칙) 같은 속성 X

3.1 The Reduction Relation

- **Reduction Relation (축소 관계)**
  - 항(term) `t` 가 `t'` 로 줄어드는 과정을 설명.
  - 등식을 이용해 한 단계로 줄어드는 것.
- **기호 체계**
  - 상수: `a, b, c, ...`
  - 함수 기호: `f, g, h, ...`
  - 변수: `x, y, z, ...`
- **예시 명세**
  - 수학적 표현:
    - f(a, g(b, x), y)=f(a, b, y)
    - h(c, c, z) = h(a, b, c)
  - Maude 코드 표현:

```
fmod M is
  sort s .
  ops a b c : → s .
  ops f h : s s s → s .
  op g : s s → s .
  vars x y z : s .
  eq f(a, g(b, x), y) = f(a, b, y) .
```

```
  eq h(c, c, z) = h(a, b, c) .
endfm
```

- **메시지**
  - 수학적 등식 집합과 Maude 모듈은 동일한 의미를 가짐.
  - 방정식 집합 E는 단순히 식의 모음이 아니라, 그것으로부터 **등식 명세 전체 (Σ,E)** 를 표현한다고 봐야 함.

3.1.1 Basic Definitions

1) Term = 트리

- **등식 속성(assoc, comm 등)이 없을 때** 항(term)은 **트리 구조**로 본다.

  예: `f(h(a,b,g(x)), f(y, f(z,b)))` → 루트가 `f`, 왼쪽 자식이 `h(...)`, 오른쪽 자식이 `f(...)`.

---

2) Position (위치) — `Pos(t)`

- 아이디어: **루트에서 자식으로 내려가는 경로를 숫자열로 표기**한다.
  - 루트 위치: **ε** (빈 문자열)
  - i번째 자식으로 내려가면 **앞에 i를 붙임** (부모 기준 자식 인덱스)
  - 예: `2.1` = "루트의 두 번째 자식 안에서 다시 첫 번째 자식"

정의 (Definition 3.1)

- 변수/상수 `t` 라면: `Pos(t) = { ε }`
- 합성항 $t = f(t_1,\dots,t_n)$ 라면:

$$Pos(t) = \{\varepsilon\} \ \cup \ \bigcup_{i=1}^{n} \{\, i.p \mid p \in Pos(t_i) \,\}$$

  - 각 인자 t_i의 위치 p 앞에 i.를 붙여서 모두 합친다.

---

3) Subterm (부분항) — `t|_p`

- 위치 `p` 가 가리키는 **부분항**.
- 정의 (Definition 3.2)
  - `t|_ε = t`
  - $f(t_1,\dots,t_n)|\_\{i.p\} = t_i|\_p$
- `p ≠ ε` 이면 **proper subterm**(진부분항).

예시 3.1 요점

- `h(a,b,g(x))` 에서
  - 위치 `3` 의 부분항은 `g(x)`
  - 위치 `3.1` 의 부분항은 `x`
- `h(a,b,g(x))` 의 부분항들: `h(a,b,g(x)), a, b, g(x), x`

(마지막 네 개는 proper subterm)

---

4) Replacement (치환 넣기) — `t[u]_p`

- 뜻: **항 `t` 에서 위치 `p` 의 부분항 `t|_p` 를 `u` 로 바꾼 결과**.
- 정의 (Definition 3.3)
    - `t[u]_ε = u`
    - $f(t_1,\dots,t_i,\dots,t_n)[u]_{i.p} = f(t_1,\dots, t_i[u]_p, \dots, t_n)$

예시 3.2 요점

- `f(a, f(x, g(y)))[b]_2 = f(a, b)`
- `f(a, f(x, g(y)))[c]_ε = c`
- `f(a, f(x, g(y)))[c]_{2.2.1} = f(a, f(x, g(c)))`

> 💡 헷갈림 방지 :
> - `t|_p` = **꺼내오기**(subterm)
> - `t[u]_p` = **바꾸기**(replace)

---

5) 변수 집합 — `vars(t)`

- 항 `t` 에 등장하는 변수들의 집합.
  예: `vars(f(a,g(x,f(b,z)))) = {x, z}`

---

6) Substitution (변수 치환) — `σ`

- 변수 → 항 매핑. 보통 $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ 로 표기.
- **동형적 확장** `σ : T_Σ(X) → T_Σ(Y)` : 항 전체에서 **모든 변수** `x` 를 **동시에** `σ(x)` 로 바꾼다.
- **postfix 표기:** `tσ` (항 `t` 에 치환 `σ` 적용)

예시 (본문)

- $\sigma = \{x \mapsto a, y \mapsto g(x,y), z \mapsto h(z,z)\}$
- `t = f(x,x,f(x,y,z))`
  ⇒ `tσ = f(a,a,f(a,g(x,y),h(z,z)))`
- **ground substitution**: 모든 변수를 **ground term(상수만으로 된 항)** 으로 치환.

---

7) Matching (패턴 매칭) — Definition 3.4

- `t` 가 `u` 와 **매치**한다 ⟺ ∃치환 `σ` s.t. **`tσ = u`**.
  이때 `u` 를 **`t` 의 instance**라고 부른다.

예시 3.3 요점

- `t = f(x,y,z)` 는 `u = f(a,g(x),h(z))` 와 매치.
  치환 $\sigma = \{x \mapsto a, y \mapsto g(x), z \mapsto h(z)\}$ 를 적용하면 `tσ = u`.

예제: `Pos` 와 `subterm` 를 끝까지 계산

대상 항

```
t = f( h(a,b,g(x)),  f(y, f(z,b)) )
```

1) 각 부분의 `Pos`

- $t = f(t_1, t_2)$
  - $t_1 = h(a,b,g(x))$
    - `Pos(a) = {ε}` → 앞에 `1.` 붙이면 `{1.1}`
    - `Pos(b) = {ε}` → `{1.2}`
    - `Pos(g(x)) = {ε, 1}` → `{1.3, 1.3.1}`
      - 루트 `{1}`
      - ⇒ $Pos(t_1) = \{1, 1.1, 1.2, 1.3, 1.3.1\}$
  - $t_2 = f(y, f(z,b))$
    - `Pos(y) = {ε}` → `{2.1}`
    - `Pos(f(z,b)) = {ε,1,2}` → `{2.2, 2.2.1, 2.2.2}`
      - 루트 `{2}`
      - ⇒ $Pos(t_2) = \{2, 2.1, 2.2, 2.2.1, 2.2.2\}$

2) 전체 `Pos(t)`

```
Pos(t) = { ε,
        1, 1.1, 1.2, 1.3, 1.3.1,
        2, 2.1, 2.2, 2.2.1, 2.2.2 }
```

3) 위치별 subterm `t|_p`

- `ε` : `f(h(a,b,g(x)), f(y,f(z,b)))`
- `1` : `h(a,b,g(x))`
- `1.1` : `a`
- `1.2` : `b`
- `1.3` : `g(x)`
- `1.3.1` : `x`
- `2` : `f(y, f(z,b))`
- `2.1` : `y`
- `2.2` : `f(z,b)`
- `2.2.1` : `z`
- `2.2.2` : `b`

💡 체크포인트

- **점(.) 앞 숫자 = 부모 기준 자식 인덱스**
- `i.p` 에서 `i` 는 "루트에서 i번째 자식으로 내려감"을 뜻함.

3.1.2 The Reduction Relation

A. 핵심 개념

- **Reduction step (축소 단계)**: 등식 `l = r` 을 항 `t` 의 부분항에 적용해, 그 부분항을 `r` 의 인스턴스로 바꾸는 한 번의 변환.
  - 예: 규칙 `g(x) = h(x)` 가 있으면
    `f(a, g(b)) → f(a, h(b))`

B. 정의 3.5 (Reduction relation)

- 표기: `t →E u` (항 t가 한 단계에서 u로 축소됨, E는 규칙 집합)
- 정의: `t →E u` 이려면 다음이 있어야 함:
  1. 규칙 `l = r` 이 E에 존재
  2. 위치 p ∈ Pos(t) 가 존재
  3. 치환 σ가 존재해서
     - `t|p = lσ` (부분항이 규칙의 왼쪽 항과 일치)
     - `u = t[rσ]p` (그 부분항을 오른쪽 항으로 교체한 결과가 u)

즉, **"부분항 매칭 → 치환 적용 → 교체"** 과정을 거친 것이 한 단계 축소.

C. 예제 3.4

규칙 집합 E = { f(x,y,z) = g(y) }

1. `f(a, b, b) → g(b)`
   - 매칭: f(x,y,z) = f(a,b,b)
   - 치환: {x ↦ a, y ↦ b, z ↦ b}
   - 결과: g(y) → g(b)
2. `h(g(b), f(a, g(x), h(z))) → h(g(b), g(g(x)))`
   - 내부 부분항 f(a, g(x), h(z))에 규칙 적용
   - 치환: {x ↦ a, y ↦ g(x), z ↦ h(z)}
   - 결과: g(y) → g(g(x))
   - 따라서 전체 항: h(g(b), f(a, g(x), h(z))) → h(g(b), g(g(x)))

3.1.3 Some Derived Relations

Reduction(축소)에서 파생된 여러 관계 정의:

1. **역관계 (inverse)**
    - `t ←E u` 는 `u →E t` 일 때 성립
2. **양방향 (either-or)**
    - `t ↔E u` 는 `t →E u` 또는 `u →E t` (혹은 둘 다)일 때 성립
3. **0회 이상 축소 (reflexive-transitive closure)**
    - 표기: `t →*E u`
    - 의미: t가 u로 0번 이상 줄어들 수 있음
    - 즉, t = u 이거나, t →E u 이거나, 또는 t →E t1 →E ... →E tn →E u
4. **경로 기반 정의 (반대 방향도 유사)**
    - `t ←*E u` 도 같은 방식으로 정의됨 (역축소를 여러 단계 거칠 수 있는 경우)
5. **1회 이상 축소 (transitive closure)**
    - 표기: `t →+E u`
    - 의미: t가 u로 1번 이상 줄어들 수 있음 (자기 자신은 제외)

Derived Relations 한눈에 정리

표기	이름	의미
`t →E u`	한 단계 축소	규칙 한 번 적용
`t ←E u`	역관계	u가 t로 축소될 때
`t ↔E u`	양방향	둘 중 하나라도 성립
`t →*E u`	0회 이상 축소	자기 자신 포함, 여러 단계 가능
`t →+E u`	1회 이상 축소	자기 자신 제외, 한 단계 이상

3.2 Operational Properties

A. 기본 용어

- **t is reducible**: 어떤 u가 있어서 `t → u` 가 가능할 때
- **t is irreducible**: 더 이상 줄일 수 없는 경우 (`t → u` 불가능)
- **u is a normal form of t**:
    - `t →* u` (0회 이상 축소로 u 도달 가능)
    - u가 irreducible
    - 만약 u가 유일한(normal form이 단 하나)라면 → `t! = u` 라고 표기 (= canonical form)
- **u is a successor of t**: `t →+ u` (t에서 한 단계 이상 줄여서 u에 도달)
- **Derivation (or reduction sequence)**:
    - 유한한 축소열: `t1 →E t2 →E ... →E tn`
    - 무한한 축소열: `t1 →E t2 →E t3 →E ...`
- **Computation in E**:

- 무한 derivation이거나
- 유한 derivation인데 마지막 항이 irreducible인 경우

B. 정의

정의 3.6 (Termination)

- 명세 E가 terminating ↔ E 안에 무한 derivation이 없음

정의 3.7 (Confluence)

- 명세 E가 confluent ↔ 임의의 항 t, t1, t2에 대해
  - `t →* t1` 이고 `t →* t2` 라면,
  - 어떤 u가 존재해서 `t1 →* u` 이고 `t2 →* u`

C. 의미

- Termination + Confluence → 계산 결과가 "어떤 순서로 규칙을 적용하든 동일"
- 즉, Maude에서 어떤 항을 줄여도 항상 같은 결과(normal form)에 도달

D. 정리 3.1

Theorem 3.1

- E가 terminating이라 하자.
- 그러면 각 항 t는 유일한 normal form을 갖는다 ↔ E가 confluent할 때, 그리고 오직 그럴 때.

Proof (정리 3.1 증명 요약)

1) "If" 방향 (Confluence ⇒ 유일한 normal form)

- E가 confluent라고 가정, 그러나 어떤 항 t가 유일한 normal form을 갖지 않는다고 가정(모순 유도).
- t가 두 개의 서로 다른 normal form u1, u2를 가진다고 하자.
  - 즉, `t →* u1` 이고 `t →* u2`.
- Confluence 정의에 따라, 어떤 u가 있어서 `u1 →* u` 그리고 `u2 →* u`.
- 하지만 u1, u2는 normal form이므로 더 이상 줄일 수 없음. 따라서 불가능.
- 모순 발생 → 각 항은 유일한 normal form을 가져야 함.

2) "Only if" 방향 (유일한 normal form ⇒ Confluence)

- 각 항이 유일한 normal form을 가진다고 가정, 그러나 E가 confluent하지 않다고 하자.
- 즉, 어떤 t, t1, t2가 있어서:
  - `t →* t1`, `t →* t2`
  - 그런데 공통 u가 없음 (`t1 →* u`, `t2 →* u` 불가)
- 하지만 각 항은 유일한 normal form을 가져야 하므로,
  - t1은 normal form t1!,
  - t2는 normal form t2!
- 만약 `t1! = t2!` 라면 그게 공통 u가 되어 confluence 성립 (가정과 모순)
- 만약 `t1! ≠ t2!` 라면 t는 두 개의 서로 다른 normal form을 갖게 됨 (가정과 모순)

- 따라서 반드시 confluent.

E. 요약

- **Termination**: 끝나는가? (무한 줄이기 없음)
- **Confluence**: 어디로 가든 같은 곳에서 만나는가? (유일한 normal form)
- **Theorem 3.1**: Termination + Confluence ⇒ 모든 항은 유일한 normal form을 가짐

3.3 Conditional Equations and Matching with `assoc / comm`

이 절에서는 조건부 방정식(conditional equations)의 연산적 의미(operational sementics)와, 연산자가 결합 법칙(associative) 혹은/그리고 교환 법칙(commuative)으로 선언된 경우, 매칭(즉, 방정식을 적용하는 과정)의 계산 복잡성에 대해 간단히 논의한다.

3.3.1 Conditional Equations

1. **조건부 방정식의 적용 방식**
    - Maude에서 조건부 방정식은

      l = r if t1 = u1 ∧ ⋯ ∧ tn = un

      형태.
    - 즉, l을 r로 바꾸기 전에, 조건 t_i = u_i 들이 모두 만족해야 함.
    - 만족 여부는 치환 σ를 적용한 뒤, **각 t_iσ와 u_iσ를 normal form까지 줄여 비교**해서 확인한다.
2. **주의점**
    - 조건부 방정식이 있는 사양(specification)은 “정의상으로는” 종료(terminating)할 수 있음.
    - 하지만 실제 Maude 실행에서는 조건 검사를 무한히 반복할 수 있음 → **운영적 종료(operational termination)가 안 될 수 있음.**
3. **예시**
    - 사양 E = {a = b if a = b}
    - 이론적으로는 종료하는 사양 (더 줄일 수 있는 항 없음).
    - 하지만 Maude에서 `red a .` 실행하면, 시스템은 “조건 a=b가 맞나?”를 계속 확인하려고 시도하면서 무한 루프에 빠짐.

3.3.2 * A-, C- and AC-matching is NP-hard

1. **배경**
    - 우리가 어떤 식(등식) l=r을 항 t에 적용하려면, l이 t의 부분항과 "매칭"되어야 함.
    - 연산자가 **결합적(Associative, A)** 이거나 **교환적(Commutative, C)** 일 경우, 매칭이 단순하지 않음.
        - 예: f(x,y)와 f(a,b) → 교환법칙 있으면 (x=a, y=b), (x=b, y=a) 두 가지 경우 다 가능.
        - 예: g(x,y)와 g(g(a,b),c) → 결합법칙 있으면 (x=g(a,b),y=c), (x=a,y=g(b,c)) 두 경우 가능.
    - 즉, **A, C, AC 연산자 때문에 매칭 경우가 폭발적으로 늘어날 수 있다.**

2. **문제의 난이도**
    - "모든 매칭 찾기"는 가능은 하지만 비효율적.
    - 심지어 "매칭이 하나라도 존재하는가?"만 확인해도 **NP-complete 문제**.
    - 따라서 이 문제를 효율적으로 푸는 일반 알고리즘은 존재하지 않는다 (단, P=NP 예외).
3. **증명 아이디어**
    - 다른 NP-complete 문제(Positive 1-in-3-SAT)를 AC-matching 문제로 변환 가능.
    - 즉, **AC-matching은 NP-complete임**을 보임.
4. **실무적 의미**
    - 이론적으로는 어렵고 어떤 경우는 지수 시간 걸림.
    - 하지만 Maude 같은 시스템에서는 **대부분의 실제 패턴에 대해 충분히 빠르게 동작하도록 최적화**해놨음.