# 8. Modeling Distributed Systems in Rewriting Logic

## 8.1 Dynamic Systems

**리라이팅 로직(Rewriting Logic)의 목적**

- **동적 시스템(dynamic systems)**을 모델링하고,
- **분산 환경에서의 동시적 변화(concurrent change)**를 분석하기 위한 논리적 프레임워크이다.
- 실제 실행은 **Maude 시스템**을 통해 가능하다.

---

**정적 시스템과 동적 시스템의 차이**

- **정적(equational) 시스템**:
  - 오직 **식(expression)**간의 **동등성(equivalence)**만 정의한다.
  - 시간이나 변화 개념이 없으며, 항상 같은 결과를 낸다.
  - 예: `2 + 1 = 3`, `length(2 5 7) = 3`.
- **동적(dynamic) 시스템**:
  - 시간이 지남에 따라 **상태(state)**가 변화한다.
  - 예: `person("Peter", 46, married)` → `person("Peter", 47, married)` → `person("Peter", 46, divorced)`
  - 변화는 대부분 **되돌릴 수 없음(irreversible)**.

---

**반응형 시스템 (Reactive System)**

- 시스템은 **환경(environment)**으로부터 입력을 받아 **상태를 바꾸거나 출력**을 생성한다.
- 예시: **운영체제(OS)**
  - `ls` → 파일 목록 출력
  - `rm` → 파일 삭제 (시스템 상태 변경)
- 특징: 비종료적(nonterminating), 비결정적(nondeterministic).
- 분석 대상: "현재 상태"와 "환경 자극에 대한 반응".

---

**분산 시스템 (Distributed System)**

- 여러 **분산된 구성 요소(component)**들이 서로 통신하며 동작.
- 예: 공유 변수(shared variable) 접근, 메시지(message) 교환.
- 각 구성 요소는 독립적이지만 상호작용을 통해 전체 시스템의 동작이 결정된다.

---

**한 줄 요약**

- 리라이팅 로직은 시스템의 상태 변화와 상호작용을 논리적으로 표현하여, 비결정적이고 동시적인 분산 시스템의 동작을 수학적으로 모델링하고 분석하는 방법이다.

### 8.1.1 Properties of Dynamic and Distributed Systems

**결정적 시스템 (Deterministic Systems)**

- **동일한 초기 상태**에서 시작하면 **항상 같은 결과**를 산출.
- 예: `3 + 2 = 5`, 정렬 프로그램 → 항상 동일한 결과.
- 함수형 Maude 모듈과 순차적 프로그램은 보통 **종료(terminating)** 해야 한다.

---

**비결정적 시스템 (Nondeterministic Systems)**

- **같은 초기 상태**에서도 **여러 가능한 경로**와 **결과**를 가질 수 있음.
- 예:
  - 사람의 상태 `person("Peter", 46, married)` → 결혼 유지, 사망, 이혼 등 여러 변화 가능.
  - 체스 게임 → 동일한 시작 위치라도 진행 방식에 따라 다른 결과 발생.
- 따라서 **결과의 예측이 불가능**하다.

---

**네트워크 및 분산 시스템의 비결정성**

- 네트워크 시스템은 본질적으로 **비결정적**이다.
- 예: 두 사용자가 동시에 항공권을 예매할 때,
  - 네트워크 부하, 라우팅 경로, 서버 상태 등에 따라 **결과가 달라짐**.
  - 다음 시도 때 결과가 바뀔 수도 있음.

---

**종료 여부 (Termination)**

- 일부 시스템(예: 인간의 생애)은 **종료(terminating)** 됨.
- 하지만 **분산 시스템(distributed systems)**은 **항상 동작(never terminating)** 해야 함.
  - 예: 운영체제, 항공 예약 시스템, 웹 서비스 등 → 지속적으로 실행되어야 함.

---

**한 줄 요약**

- 결정적 시스템은 항상 같은 결과를 내지만, 동적/분산 시스템은 환경과 상호작용 속에서 비결정적이며, 대부분 종료되지 않고 계속 동작한다.

## 8.1.2 Behaviors of Distributed Systems

**인터리빙 의미론 (Interleaving Semantics)**

- 시스템 내 **하나의 이벤트만 동시에 발생**할 수 있다고 가정하는 모델.
- 여러 구성요소(component)가 순차적으로 작업을 수행할 때, **모든 가능한 실행 순서(interleaved order)** 를 고려해야 함.
- 예: 세 구성요소 c_1, c_2, c_3가 각각 4개의 연산을 수행한다면, 가능한 실행 순서는 무려 **34,650가지**나 존재.

---

**분석의 복잡성**

- 단순한 시스템조차 수만 개의 가능한 실행 경로를 가지므로, **분산 시스템의 이해와 분석은 매우 어렵다.**
- 모든 구성요소가 언제 실행될 수 있는지를 고려해야 하며, 실행 순서가 달라지면 결과도 달라질 수 있다.

---

**인터리빙 모델의 적용**

- 한 번에 **하나의 구성요소만 실행** 가능한 시스템에 적합.
  - 예: 여러 프로세스가 **공유 자원(shared resource)**을 사용하는 경우.
  - 서버가 요청을 순차적으로 처리하거나, 하나의 데이터에 접근하는 상황.

---

**동시 실행 (Concurrent Execution)**

- **분산 시스템**에서는 두 개 이상의 구성요소가 **동시에(concurrently)** 실행될 수 있다.
- 예:
  - 세 구성요소가 동시에 첫 번째 연산을 수행.
  - c_1과 c_3가 각각 별도의 컴퓨터에서 두 번째 연산을 동시에 수행.
- 단, **서로 다른 컴퓨터**에서 실행되며 **공유 자원에 접근하지 않아야 함.**

---

**한 줄 요약**

- 인터리빙 모델은 분산 시스템의 가능한 모든 실행 순서를 기술하지만, 실제 분산 환경에서는 여러 구성요소가 동시에 실행될 수 있어 분석 복잡성이 급격히 증가한다.

## 8.2 Modeling Dynamic Systems in Rewriting Logic

**등식 명세(equational specification)의 한계**

- 동적 시스템을 표현하기에는 부적절함. 이유는 다음과 같다:
  - (i) 등식 논리에서는 **변화가 가역적(reversible)**이므로 시간의 흐름을 표현할 수 없음.
  - (ii) 등식 명세는 **항상 수렴(confluent)** 해야 하지만, 동적 시스템은 종종 **비결정적 (nondeterministic)**이라 수렴하지 않음.
  - (iii) 등식 명세는 **종료(terminating)** 되어야 하나, 많은 동적 시스템은 **지속적(non-terminating)**으로 동작해야 함.

**리라이팅 로직(Rewriting Logic)의 필요성**

- 예:
  - 단순히 `person(X, N, S) = person(X, N + 1, S)` 로 나이를 모델링하면 `person("Peter", 46, married) = person("Peter", 20, married)` 와 같은 **비현실적인 결론**이 도출됨.
- 따라서, **리라이팅 로직(rewriting logic)**을 통해 변화가 일어나는 시스템의 상태 전이를 명확히 표현해야 함.

---

### 8.2.1 Rewrite Rules

**리라이트 규칙의 정의**

- 리라이팅 로직에서 **동적 행위(dynamic behavior)**는 **리라이트 규칙(rewrite rule)**로 모델링된다.
- 각 규칙은 시스템의 **지역적 상태 전이(local transition)**를 정의함.
- 예시 (“생일” 행동):

$$person(X, N, S) \rightarrow person(X, N + 1, S)$$

---

**시스템 상태 변화의 표현**

- 시스템의 상태 변화는 다음과 같이 표현됨:

$$t \to t'$$

  → 상태 t가 리라이트 규칙을 0회 이상 적용해 **다른 상태 t'**로 진화할 수 있음을 의미.

- 예시:
  - 가능: `person("Peter", 46, married) → person("Peter", 56, married)`
  - 불가능: `person("Peter", 46, married) → person("Peter", 20, married)`

---

**레이블(Label)**

- 리라이트 규칙은 상태 변화를 유발하는 **행동(action)** 또는 **이벤트(event)**의 이름을 나타내는 **레이블(label)**을 가질 수 있음.
- 예:

$$birthday : person(X, N, S) \to person(X, N + 1, S)$$

- 레이블은 Maude의 실제 계산에는 영향을 미치지 않음.

---

**등식적 데이터와 리라이팅**

- 시스템의 데이터는 여전히 **등식적(equational)**으로 정의됨.
- 예: `+` 연산은 내장된 함수로 취급됨.
- 따라서:

$$person(\text{"}Roland\text{"}, 7, single) \to person(\text{"}Roland\text{"}, 8, single)$$

  은 한 단계에서 발생하며, `person("Roland", 7 + 1, single)` 과 **동치(equivalent)** 임.

---

**한 줄 요약**

- 8.2: 등식 논리로는 변화와 비결정성을 표현하기 어렵기 때문에, 리라이팅 로직이 필요하다.
- **8.2.1:** 리라이팅 로직에서는 시스템의 상태 전이를 리라이트 규칙으로 정의하며, 각 규칙은 행동(이벤트)에 의해 상태가 진화하는 과정을 수학적으로 기술한다.

## 8.2.2 Rewriting Logic Specifications

**개념 요약**

- **리라이팅 로직 명세(rewriting logic specification)**는 등식 논리 명세(equational specification)에 **레이블이 붙은 리라이트 규칙(labeled rewrite rules)**을 확장한 형태이다.
- 이러한 규칙은 시스템의 **상태 변화(state change)**, 즉 **지역적 전이(local transition)**를 기술한다.

---

**정식 정의**

리라이팅 로직 명세는 다음과 같은 튜플로 정의된다.

$$\mathcal{R} = (\Sigma, E, L, R)$$

- **Σ**: 대수적 시그니처 (algebraic signature)
- **E**: 등식(equations)과 멤버십 공리(membership axioms)의 집합
- **L**: 레이블(labels)의 집합
- **R**: 리라이트 규칙(rewrite rules)의 집합

리라이트 규칙은 다음 두 가지 형태를 가진다:

- **무조건적 규칙:** l : t → t′
- **조건적 규칙:** l : t → t′ if cond
  - `cond`는 등식, 멤버십, 혹은 다른 리라이트 조건의 결합(conjunction)으로 구성됨.

---

**Maude에서의 표현**

Maude에서는 리라이트 이론을 **system module**로 선언하며, 함수형 모듈(`fmod`) 대신 `mod` ~ `endm` 키워드를 사용한다.

규칙 표현 방식:

```
rl [label] : t => t' .
crl [label] : t => t' if cond .
```

- 조건 `cond`에는 Bool 값, 등식, 멤버십 검사, 리라이트 조건(`u => u'`) 등이 포함될 수 있음.
- 리라이트 조건은 0회 이상 리라이트 단계를 거쳐 만족될 수 있다.

---

**비결정성 (Nondeterminism)**

- 많은 동적 시스템은 경쟁 상태(race condition) 등으로 인해 **비결정적**이다.
- 등식 논리로는 비결정성을 표현할 수 없지만, **리라이팅 로직**에서는 **하나의 상태에 여러 리라이트 규칙을 적용**하여 이를 쉽게 표현할 수 있다.

---

**예시**

**비결정적 선택 연산자** `_?_` 의 예:

```
mod CHOICE-INT is including INT .
  op _?_ : Int Int -> Int [ctor] .
  vars I J : Int .
  rl [choose_first]  : I ? J => I .
  rl [choose_second] : I ? J => J .
endm
```

- 항 `3 ? 5` 는 **3 또는 5** 중 하나로 변화할 수 있음.

이처럼 리라이팅 로직은

- **비결정성(nondeterminism)**

- **비수렴성(non-confluence)**
- **비종료성(non-termination)**

  을 자연스럽게 표현할 수 있다.

---

**한 줄 요약:**

- Rewriting Logic Specification은 등식 논리를 확장하여 시스템의 상태 변화를 규칙적으로 기술하는 모델이며, Maude에서는 rl과 crl 규칙으로 구현된다. 이를 통해 비결정적/비수렴적 시스템의 동작을 자연스럽게 표현할 수 있다.

## 8.2.3 Examples

**Football Game 시뮬레이션**

- **목적:** 미식축구(American football) 경기의 점수 변화를 시뮬레이션.
- **상태 표현:**

  `"Steelers" vs "Patriots" 35 : 0`

  → 홈팀("Steelers"), 원정팀("Patriots"), 점수(35:0).
- **리라이트 규칙:**
  - 홈팀 득점 규칙: 터치다운(+6), 필드골(+3), 추가점(+1), 2점 전환(+2), 세이프티(+2)
  - 원정팀 득점 규칙: 동일한 구조로 원정팀 점수를 증가시킴.
- **핵심 요약:**
  - 각 득점 이벤트를 리라이트 규칙으로 정의하여 점수 변화를 자동으로 모델링한다.

---

**Modeling the Life of a Person**

- **목적:** 사람의 생애 변화를 상태 전이로 모델링.
- **상태 표현:**

  `person(name, age, status)`
  - `status` : single, engaged, married, separated, divorced, deceased 등.
- **리라이트 규칙:**
  - `birthday` : 나이 증가 ( `N + 1` ), 단 사망 상태가 아니어야 함.
  - `successful-proposal` : 만 15세 이상이면서 미혼/이혼 상태 → engaged
  - `marriage` : engaged → married
- **핵심 요약:**
  - 사람의 나이 및 결혼 상태 변화를 조건적 리라이트 규칙으로 정의함.

---

**Coffee Bean Game**

- **목적:** 단순한 1인용 게임을 리라이팅 로직으로 모델링.
- **규칙:**
  - 검은 콩 2개 → 흰 콩 1개로 대체
  - 흰 콩과 검은 콩이 인접할 경우 → 검은 콩 제거

- **목표:** 가능한 한 **콩의 개수를 최소화**하는 것.
- **핵심 요약:**

  단순 규칙 기반의 퍼즐도 리라이팅 로직을 통해 상태 변화로 자연스럽게 모델링 가능함.

---

**한 줄 요약:**

- 이 절은 리라이팅 로직을 이용해 실제 시스템(경기, 사람의 생애, 퍼즐 등)의 상태 변화를 규칙으로 표현하는 방법을 보여주며, 복잡하거나 비결정적인 동작을 논리적으로 모델링할 수 있음을 설명한다.

## 8.3 Concurrency

- 동시성은 **여러 개의 독립적인 rewrite 규칙**이 서로 **간섭하지 않고 동시에 수행될 수 있음**을 의미한다.
- 즉, 두 개의 상태 전이가 서로 영향을 미치지 않는다면 하나의 "병렬적인 rewrite step"으로 합쳐질 수 있다.
- 이 개념은 시스템 전체의 행동을 더 자연스럽고 효율적으로 모델링하는 기반이 된다.

### 8.3.1 Sideways Concurrency (병렬 동시성)

- 예제에서는 **인구(population)**이 여러 `person` 객체로 구성된 multiset으로 표현된다.
- 각 사람은 이름, 나이, 상태(single, married, engaged 등)를 가진다.
- 다음과 같은 규칙들이 존재한다:
  - **birthday**: 나이를 1 증가시킴
  - **engagement**: 두 명의 single/divorced 성인이 서로 약혼
  - **wedding**: 두 명의 약혼 상태를 결혼 상태로 변경

**핵심 개념:**

- 서로 다른 사람들(`Hamlet`, `Ophelia`)의 생일이 **동시에 일어날 수 있음** → 두 개의 rewrite step이 병렬로 수행됨.
- 동시에 두 쌍의 약혼(`Hamlet–Rosencrantz`, `Ophelia–Juliet`)도 가능하지만,
  - **한 사람이 두 번의 약혼에 동시에 포함되는 것은 불가능함** (논리적 충돌 방지).
- 따라서 "모든 요소가 동시에 활동해야 하는 것은 아니다" - 일부만 rewrite에 참여할 수 있다.

### 8.3.2 Nested Concurrency (중첩 동시성)

- 리라이트 규칙 I: f(x) → g(x)가 있을 때, 다른 프로세서가 x의 내부에서 또 다른 rewrite를 수행할 수 있다면, 두 프로세서가 **중첩된 형태로 동시에 작업**하는 것이 가능하다.
- 즉, 한 프로세서가 "바깥"의 구조 f → g를 처리하고, 다른 프로세서가 "안쪽"의 구조 x → x'를 동시에 처리하는 것.
- 이런 식으로 **작업을 위임(delegate)** 하는 구조를 **Nested Concurrency (중첩 동시성)** 이라고 부른다.

## 8.4 Deduction in Rewriting Logic

**핵심 개념**

- 이 절에서는 **Rewriting Logic** 내에서 **rewrite 관계**와 **concurrent rewrite**의 개념을 **형식적으로 정의**한다.
  - 여기서는 **조건 없는(conditional-free)**, **단일 정렬(one-sorted)** 명세만 고려한다.

---

**Rewriting Logic의 기본 구조**

- Rewriting Logic 명세는

$$\mathcal{R} = (\Sigma, E, L, R)$$

  로 정의되며, **논리식(sequent)**은 t → u 형태를 갖는다:
- 이는 "규칙들을 사용하여 상태 t에서 u로 도달 가능하다"는 의미이다.
- 즉, t로부터 u로 0회 이상의 rewrite를 통해 도달할 수 있음을 뜻한다.

---

**주요 표기법**

- t(x_1, ..., x_n): 항 t의 변수들을 명시적으로 표현.
- t(u_1/x_1, ..., u_n/x_n): t 안의 각 변수 x_i를 u_i로 치환한 결과.
- 예:

  t = f(g(x), h(a, y)) 일 때, t(g(y)/x, a/y) = f(g(g(y)), h(a, a))

---

**Deduction Rules (추론 규칙)**

- Rewrite 관계 t → u가 성립한다는 것은 아래 규칙들을 **유한 횟수로 적용**하여 얻을 수 있음을 의미한다.

| 규칙 | 설명 |
|---|---|
| **Reflexivity** | 모든 항 $t$에 대해 $t \to t$ 성립 |
| **Equality** | 등식 $E$에 의해 동등한 항끼리 rewrite 가능 |
| **Congruence** | 함수 기호 $f$ 안의 인자들이 rewrite되면, 전체 $f(t_1, \ldots, t_n)$도 rewrite됨 |
| **Replacement** | rewrite 규칙 $l : t(x_1, \ldots, x_n) \to u(x_1, \ldots, x_n)$이 있을 때, $t_i \to u_i$가 모두 성립하면 $t(t_i/x_i) \to u(u_i/x_i)$도 성립 |
| **Transitivity** | $t_1 \to t_2$ 및 $t_2 \to t_3$이면 $t_1 \to t_3$도 성립 |

- 즉, rewriting logic의 추론 규칙은 **등식 논리의 규칙과 매우 유사**하며, 단 **대칭성(Symmetry)**만 없다.

---

**Proposition 8.1**

> 등식 명세 $(\Sigma, E)$는 rewrite 규칙 집합으로 변환될 수 있다.
> 즉, 등식적 변환($t \leftrightsquigarrow_E^* u$)은 rewrite 논리에서
>
> $$(\Sigma, \emptyset, \{l\}, rules(E)) \vdash t \to u$$
>
> 와 동치이다.

---

**Corollary 8.1**

- 일반적으로, 주어진 rewriting logic 명세 R 안에서 어떤 항 t가 u로 rewrite 가능한지를 **결정하는 것은 불가능(undecidable)** 하다.

## 8.4.1 Concurrent Steps

**핵심 개념**

- **Concurrent rewrite**란 여러 **rewrite rule**이 **한 번에(one step)** 적용되는 과정이다.
- "한 단계(one step)"에서는 **Transitivity(전이성)**을 사용하지 않는다.

---

**주요 규칙**

1. **Congruence rule (병렬적 concurrency)**
   - 서로 독립적인 두 rewrite가 동시에 일어남.
   - 예: a → b, c → d가 각각 한 단계에서 가능하다면 f(a, c) → f(b, d)도 한 concurrent step으로 가능.
2. **Replacement rule (중첩 concurrency)**
   - 외부 함수가 규칙을 적용하면서 내부의 여러 변수가 동시에 rewrite됨.
   - 예: l: f(x, y) → g(x, y) / a → b, c → d

     ⇒ f(a, c) → g(b, d) (하나의 nested concurrent step)

---

**정의 (Definition 8.3)**

- **Concurrent rewrite step**:

  Reflexivity, Equality, Congruence, Replacement 만 사용하여 유도된 한 단계 rewrite. (Transitivity 사용 불가)
- **Sequential rewrite step**:

  Replacement rule이 정확히 **한 번** 사용되는 한 단계 rewrite. → 즉, "규칙이 실제로 적용되는 순간"을 의미함.

---

**예시 요약**

> **Example 8.5**
> 규칙
> $l_1 : f(x) \to g(x), l_2 : a \to b$
> → $f(f(f(a))) \to g(g(g(b)))$ 는 one-step concurrent rewrite
> (Transitivity 사용 안 함, Replacement 3번 적용)
>
> **Example 8.6**
> $h(g(a,b), g(c,d), g(e,f)) \to h(g(a',b'), g(c',d'), g(e',f'))$
> Replacement + Congruence만 사용한 concurrent rewrite.
> → 동시에 여러 부분이 rewrite됨.
> → Sequential rewrite는 각 부분을 순차적으로 rewrite한 경우.

---

**Proposition 8.2 (Sequentializability)**

- 모든 concurrent rewrite는 결국 여러 sequential step들의 연쇄로 분해할 수 있다.

$$t \to t_1 \to \ldots \to t_n \to t'$$

- 즉, **병렬로 일어난 rewrite도 순차적 rewrite의 조합으로 해석 가능.**

---

**Behavior / Run**

- 여러 one-step rewrites의 연속(유한 또는 무한)을 **시스템의 행동(behavior) 또는 실행(run)** 이라 부름.

---

**Example 8.7 (swap 예제)**

Maude 명세:

```
crl [swap] : I L J ⇒ J L I if J < I .
```

- 리스트 내에서 swap 규칙 반복 적용 → 정렬됨.
- 예: `0 3 2 1 → 0 1 2 3` 은 **하나의 concurrent step**으로 가능.

  (0→0, 3→3, 2→2, 1→1은 Reflexivity / Replacement / Congruence 로 증명)

## 8.4.2 Termination and Confluence

| 개념 | 의미 | 조건 |
|---|---|---|
| **Termination (종결성)** | Rewrite 과정이 **무한히 반복되지 않음** | 무한한 one-step rewrite 시퀀스가 존재하지 않아야 함 |
| **Confluence (합류성)** | 서로 다른 경로로 rewrite돼도 **같은 결과로 합쳐짐** | t → t_1, t → t_2 일 때, 어떤 u가 존재하여 t_1 → u, t_2 → u |

## 8.5 Frozen Operators *

**핵심 개념**

- **Frozen operator**는 rewriting logic에서 **Congruence 규칙이 적용되지 않는 연산자**이다.
- 즉, 내부 인자의 rewrite가 연산자 외부로 **전파되지 않도록(freeze)** 하는 장치.

---

**배경**

- 예제 `SORT` 모듈에서

  ```
  op first : List → Int .
  ```

  를 정의하면, 리스트의 첫 원소를 반환하는 함수가 된다.
- 그런데 rewrite (5 2 → 2 5)가 존재하면

  → Congruence 규칙에 의해 first(5 2) → first(2 5)

  → Equality 규칙에 의해 (5 → 2) 로 귀결됨.

  **⇒ 의도치 않은 rewrite 발생.**

---

**해결: frozen 선언**

```
op first : List -> Int [frozen] .
```

- 이렇게 하면 rewrite (t -> t')가 있어도 first(t) -> first(t')는 **허용되지 않는다.** (Congruence rule 제외)
- 필요 시, 인자 중 **일부만 frozen**으로 지정할 수도 있다. (예: 두 번째 인자만 freeze)\

## 8.6 Denotational Semantics (표현적 의미론) *

**핵심 개념**

- 등식적 명세(equational specification)의 모델:

  **initial algebra (초기 대수)** - 모든 식이 "동일 원소" 또는 "무관한 원소"로 평가됨.
- 하지만 **rewrite theory**에서는 서로 다른 항 t, t'이 rewrite (t -> t')로 **연결될 수 있음.**

  따라서 rewrite 논리의 모델은 **대수(algebra)**가 아니라 **범주(category)** 로 해석해야 함.

---

**Category 정의 (Definition 8.4)**

한 범주 $\mathcal{A} = (A, M)$ 은

- **A:** 객체(object)들의 집합
- **M:** 화살표(morphism, arrow)들의 집합 $f : A \to B$

이며 다음 조건을 만족한다:

1. **합성(Composition):**
   $f : A \to B, g : B \to C \to (g \circ f : A \to C)$
2. **항등(Identity):**
   각 객체 A에 대해 $id_A : A \to A$ 존재,
   $id_A \circ f = f, g \circ id_A = g$
3. **결합성(Associativity):**
   $(f \circ g) \circ h = f \circ (g \circ h)$

---

**Rewrite Theory의 모델**

- **객체(Object):** 초기 대수의 원소들
- **화살표(Morphism):** rewrite 관계 (t -> t')
- **Reflexivity** -> 자기 자신으로 향하는 화살표 존재
- **Transitivity** -> 화살표 합성 가능

즉, rewrite theory의 초기 모델은 "객체(식)"와 "화살표(변환)"로 구성된 **범주(category)**로 표현된다.