3. Theories

3.1 EUF : Equality and Uninterpreted Functions

EUF 기본

- **EUF**(Equality with Uninterpreted Functions) : 등식 + 해석되지 않은 함수만 다루는 1차 논리.
- **결정 절차(quantifier-free)** : 등식들로 항을 묶는 **동치류(equivalence classes)**를 **Union-Find**로 관리.

불일치식(≠) 판정 아이디어

- `a ≠ d`의 만족 가능성은 **a와 d가 같은 동치류인지**만 보면 됨.
    - 다른 동치류 → 모순 없음(sat)
    - 같은 동치류 → 모순(unsat)

함수가 있을 때: 합동 규칙 필요

- **합동 규칙(Confruence rule)** : 인수가 같으면 함수 값도 같다
  $x_1 = y_1, \ldots, x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$
- 함수가 들어오면 단순 Union-Find만으로 부족 → **Congruence Closure**로 확장.

교재 예제 흐름

- 가정된 등식/부등식
    - 등식: `a = b, b = c, b = s` → 동치류 `{a, b, c, s}`
      등식: `d = e, d = t` → 동치류 `{d, e, t}`
    - 부등식: `f(a, g(d)) ≠ f(b, g(e))`
    - 부분항 기호화:
      $v_1 := g(e)$, $v_2 := g(d)$, $v_3 := f(a, v_2)$, $v_4 := f(b, v_1)$
- Bottom-up 합동 닫힘
    1. `d = e` 적용 → **g(d) = g(e)** ⇒ $v_2 = v_1$ (동치류 병합: $\{v_1,v_2\}$)
    2. `a = b` 및 $v_2 = v_1$ 적용 → $\mathbf{f(a,v_2) = f(b,v_1)}$ ⇒ $v_3 = v_4$ (병합: $\{v_3,v_4\}$)
- 귀결
    - 위 추론으로 v_3와 v_4는 동일.
    - 초기 조건은 `f(a,g(d)) ≠ f(b,g(e))` (즉 $v_3 \neq v_4$).
    - **모순 → 공식은 unsat(만족 불가능)**.

3.1.1 Congruence Closure (합동 닫힘)

개념

- 합동 닫힘 : "같다" 관계를 **최대한 자세히** 퍼뜨려서, 누가 누구랑 같은지 **묶음(동치류)**을 만드는 방법.
- `T` : terms의 집합, `E` : 등식(=)들의 집합.

- **Congruence closure** `cc` : `T` 를 **가장 잘게** 나누는(= *finest partition*) 동치 분할로서,
  1. `E` 에 들어있는 등식은 반드시 같은 묶음(동치류)에 넣고,
  2. 같은 함수에 자리별 인수가 각각 같은 묶음이면 결과도 같은 묶음에 넣는다 *(합동성: inputs equal ⇒ outputs equal)*.

형식 정의 (핵심 규칙)

- 만약 $(s = t) \in E$ 이면, s 와 t 는 `cc` 에서 **같은 동치류**.
- $s := f(s_1,\dots,s_k)$, $t := f(t_1,\dots,t_k)$ 가 있을 때,
  - 모든 i 에 대해 s_i 와 t_i 가 `cc` 에서 **같은 동치류**라면,
  - s 와 t 도 `cc` 에서 **같은 동치류**.
- 표기: `cc : T → 2^T` (각 term을 그 term이 속한 **동치류**로 매핑).

직관

- 등식으로 "같다"는 것들을 **묶음(동치류)**으로 만든다.
- 함수 `f` 에 대해 **자리별로 같은 묶음**이면 **결과도 같은 묶음**으로 강제한다.
- "finest(가장 잘게)"는 위 두 규칙을 지키면서 **불필요하게 합치지 않은** 최소한의 묶음을 의미.

아주 작은 예

- $E = \{ a = b, d = e \}$
- 합동성 적용:
  - $g(d)$ 와 $g(e)$ 는 같은 묶음.
  - $a = b$ 이고 $g(d) = g(e)$ 이면 $f(a, g(d))$ 와 $f(b, g(e))$ 도 같은 묶음.
- 따라서 $f(a, g(d)) \neq f(b, g(e))$ 를 추가하면 **모순(unsat)**.

왜 중요?

- EUF에서 **만족 가능성**을 빠르게 판단하기 위한 표준 방식.
- 실제 구현은 보통 Union-Find + 합동성 전파로 동치류를 유지/병합한다.

3.1.2 EUF Models

앞서의 예제에서 만족 가능한(satisfiable) 버전은 다음과 같다.

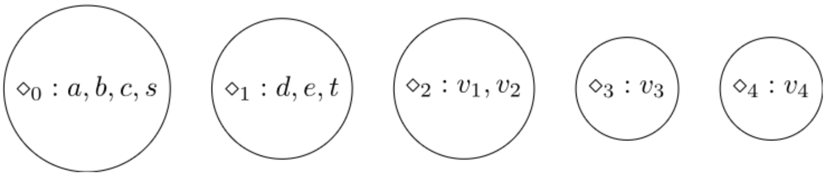
$$a = b, b = c, d = e, b = s, d = t, f(a, g(d)) \neq f(g(e), b)$$

이것은 다음의 정의와 등식을 유도한다.

$$a = b, b = c, d = e, b = s, d = t, v_3 \neq v_4$$
$$v_1 := g(e), v_2 := g(d), v_3 := f(a, v_2), v_4 := f(b, v_1)$$

각 동치류에 서로 다른 값을 연결할 수 있다.

동치류 집합은 다음과 같다.



이 공식을 Z3에 제시하면 다음과 같이 표현할 수 있다.

```
S = DeclareSort('S')
a, b, c, d, e, s, t = Consts('a b c d e s t', S)
f = Function('f', S, S, S)
g = Function('g', S, S)
solve([a == b, b == c, d == e, b == s,
       d == t, f(a, g(d)) != f(g(e), b)])
```

이를 실행하면 Z3는 다음과 같은 모델을 생성한다.

```
[s = S!val!0, b = S!val!0, a = S!val!0,
 c = S!val!0, d = S!val!1, e = S!val!1, t = S!val!1,
 f = [(S!val!2, S!val!0) → S!val!4, else → S!val!3],
 g = [else → S!val!2]]
```

이 모델에서 `S!val!0` 은 `S!val!1` 과 구별되는 새로운 상수(fresh constant)이다.

함수 `f` 의 그래프는 `(S!val!2, S!val!0)` 을 `S!val!4` 로 대응시킨다.

그 외의 모든 인자는 `else` 절에 의해 `S!val!3` 으로 매핑된다.

`else` 절은 모델에서 명시적으로 나열되지 않은 인자 조합에 대한 **기본(default) 해석**으로 사용된다.

`S` 의 해석은 유한한 집합으로 다음과 같이 표현된다.

```
{S!val!0, S!val!1, S!val!2, S!val!3, S!val!4}
```

핵심 요약

EUF 모델은 등식/부등식 관계를 기반으로 각 항(term)을 등가 클래스에 묶고, 각 클래스에 서로 다른 해석값을 부여하여 식을 만족시키는 모델을 구성한다.

> 💡 EUF는 함수의 실제 의미를 몰라도 "같음 관계"만으로 논리식을 검증할 수 있는 이론이며, Z3는 이를 위해 각 항을 등치류로 묶고, 각 그룹에 다른 해석값을 부여해 모델(EUF Model)을 구성한다.

3.2 Arithmetic (산술)

3.2.1 Solving LRA : Linear Real Arithmetic

무엇을 하려는 건가?

Z3는 이런 **실수(real number) 관련 부등식** 문제를 풀려고 함:

```
x, y = Reals('x y')
solve([x >= 0, Or(x + y <= 2, x + 2*y >= 6),
       Or(x + y >= 2, x + 2*y > 4)])
```

즉, "x, y가 어떤 값을 가지면 이 식이 참이 될까?" 를 찾는 것.

Z3의 내부 처리 아이디어

Z3는 이걸 단순히 계산으로 푸는 게 아니라, **'표(tableau)'라는 구조(=일종의 표 계산)**를 만들어서 **변수 관계를 정리**함.

예를 들어 새 변수 `s₁, s₂` 를 만들어서

$$s_1 = x + y$$
$$s_2 = x + 2y$$

로 표현하고, 부등식은

$$x \geq 0,\ (s_1 \leq 2 \lor s_2 \geq 6),\ (s_1 \geq 2 \lor s_2 > 4)$$

이런 식으로 단순화해서 다룸.

왜 이런 짓을 하냐?

이렇게 하면 식 전체를 **"Ax = 0" 꼴** (즉, 선형방정식)로 만들 수 있고, 이걸 **Simplex 알고리즘**으로 효율적으로 풀 수 있기 때문임.

- Ax = 0 꼴 : s1 - x - y = 0, s2 - x - 2y = 0

Simplex는 "선형 부등식 안에서 가능한 값들을 찾는 알고리즘"임.

핵심 아이디어

- `s₁, s₂` 는 **기초 변수(basic)** → 다른 변수로부터 결정됨
- `x, y` 는 **비기초 변수(non-basic)** → 자유롭게 조정 가능
- 계산 중 "pivot"이라는 과정을 통해 변수 역할을 바꾸면서 제약조건을 만족하는 조합을 찾음.

예시로 정리

초기 조건을 이렇게 둔다고 하자.

$$x = y = s_1 = s_2 = 0$$
$$x \geq 0,\ s_1 \leq 2,\ s_1 \geq 2$$

그럼 $s_1 = 2$로 맞추고, 식을 정리하면,

$$y + x - s_1 = 0$$
$$s_2 - x - 2s_1 = 0 \text{ (왜? s2 - x - 2y = 0에 } y = s_1 - x \text{ 대입)}$$

이때 계산 결과로 얻은 값은

```
x = 0, s₁ = 2, s₂ = 4, y = 2
```

이 조합이 원래 식을 만족함 (즉, **모델**).

한 줄 요약

Z3는 실수 방정식을 Simplex 테이블로 변환해 변수들 간의 선형관계를 정리하면서 조건을 만족하는 값(x, y 등)을 효율적으로 찾아낸다.

3.2.2 Solving Arithmetical Fragments

개념 요약

Z3는 **산술 제약식(Arithmetic Constraints)**을 다루는 여러 종류의 **논리 조각(fragment)**을 지원한다.

각 fragment마다 사용하는 **결정 절차(decision procedure)** 와 **알고리즘**이 다르다.

주요 산술 논리와 해결 방법 (Table 1 요약)

Logic	설명	Solver	예시
LRA	Linear Real Arithmetic (선형 실수 산술)	Dual Simplex	`x + ½y ≤ 3`
LIA	Linear Integer Arithmetic (선형 정수 산술)	Cuts + Branch	`a + 3b ≤ 3`
LIRA	Mixed Real/Integer (실수+정수 혼합)	다양한 혼합 기법	`x + a ≥ 4`
IDL	Integer Difference Logic (정수 차 논리)	Floyd–Warshall	`a − b ≤ 4`
RDL	Real Difference Logic (실수 차 논리)	Bellman–Ford	`x − y ≤ 4`
UTVPI	Unit Two-Variable Per Inequality (단위계수 2변수 부등식)	Bellman–Ford	`x + y ≤ 4`
NRA	Non-linear Real Arithmetic (비선형 실수 산술)	Model-based CAD	`x² + y² < 1`
NIA	Non-linear Integer Arithmetic (비선형 정수 산술)	CAD + Branch / Linearization	`a² = 2`

- Z3는 문제 형태에 따라 **Simplex**, **Bellman–Ford**, **CAD**, **Branch-and-Bound** 등의 알고리즘을 자동으로 선택하여 풀이함.

Z3가 아직 완전히 다루지 못하는 Fragment들 (Table 2)

Fragment	예시
Horn Linear Real Arithmetic	`3y + z − ½x ≤ 1`
At most one variable is positive	–
Two-variable per inequality	`3x + 2y ≥ 1`
Min-Horn	`x ≥ min(2y + 1, z)`
Bi-linear arithmetic	`3xx' + 2yy' ≥ 2`
Transcendental functions	`e⁻ˣ ≥ y`
Modular linear arithmetic	`a + 3b + 2 ≡ 0 (mod 5)`

- 지수함수, 모듈러(mod) 연산, 비선형항 등은 Z3가 완벽히 지원하지 않음.

Z3의 산술 처리 특징

- Z3는 **무한 정밀도 산술(infinite precision arithmetic)**을 사용함.
  → 정수와 유리수를 **반올림 없이 정확하게 표현**함.
- 장점: 계산 결과의 **정확성(safety)** 보장
- 단점:
  - 숫자가 커질수록 연산량 급증
  - 큰 계수나 긴 소수 표현이 있는 식은 **성능 저하** 발생 가능

한 줄 요약

Z3는 다양한 산술 논리 조각별로 최적화된 알고리즘(Simplex, Bellman-Ford, CAD 등)을 사용하며, 무한 정밀 산술을 통해 정확한 해를 보장하지만, 큰 수나 복잡한 식에서는 계산 시간이 늘어날 수 있다.

3.3 Arrays

개념 요약

- Z3에서 배열은 **함수(Function Space)**로 표현된다.
  → 즉, "인덱스(index) → 값(value)"의 관계를 갖는 **함수형 객체**임.
- 예시 선언:

```
A = Array('A', IntSort(), IntSort())
```

  → 정수를 정수로 매핑하는 배열 `A` 생성.

배열 제약(Constraints) 예시

```
solve(A[x] == x, Store(A, x, y) == A)
```

- `Store(A, x, y)` : 배열 A의 x번째 값을 y로 갱신한 새 배열
- 위 식이 참이 되려면 반드시 `x == y` 여야 함.
  → 따라서 Z3는 **x와 y가 같음**을 모델로 반환.

Lambda 표현

Z3는 배열을 함수로 다루므로, 함수 `f(x, y)` 는 **람다(lambda)** 를 사용해 배열 형태로 변환할 수 있다.

```
Lambda([x, y], f(x, y))
```

- 만약 `f` 의 타입이 `A × B → C` 라면 `Lambda([x, y], f(x, y))` 의 타입은 `Array(A, B, C)` 가 됨.

주요 내장 함수 (Array 연산)

연산	설명	람다 표현식
`a[i]`	배열 `a` 의 `i` 번째 원소 선택	`Select(a, i)`
`Store(a, i, v)`	배열 `a` 의 `i` 번째를 `v` 로 갱신한 새 배열	`Lambda(j, If(i == j, v, a[j]))`
`K(D, v)`	모든 인덱스에 값 `v` 가 들어있는 상수 배열	`Lambda(j, v)`

연산	설명	람다 표현식
`Map(f, a)`	배열 `a` 의 각 원소에 함수 `f` 를 적용	`Lambda(j, f(a[j]))`
`Ext(a, b)`	배열의 **확장성(Extensionality)** : 두 배열이 같으려면 모든 인덱스의 값이 같아야 함	`Implies(a[Ext(a, b)] == b[Ext(a, b)], a == b)`

핵심 요약

Z3의 배열은 '함수형 객체(Function Space)'로 처리되며, **배열의 수정(Store), 선택(Select), 맵(Map) 등이 모두 람다(Lambda) 함수로 표현된다.**

3.3.1 Deciding Arrays by Reduction to EUF (배열을 EUF로 환원하여 결정하기)

핵심 아이디어

Z3는 `Store` , `K` , `Map` , `Ext` 등의 배열 연산이 포함된 식을 **EUF(Equality with Uninterpreted Functions)** 형태로 변환해 해결한다.

즉, "배열 문제를 함수(=) 관계 문제로 바꿔서" 푼다.

Store 연산 변환

`Store(a, i, v)` 는 배열 `a` 의 `i` 번째를 `v` 로 바꾼 새 배열이므로,

다음 두 제약으로 바꿔서 EUF로 표현함

```
s.add(Store(a, i, v)[j] == If(i == j, v, a[j]))
# Store는 기존 배열 a를 직접 바꾸는 게 아니라, 새로운 배열을 만들어 반환하는 것임
# 따라서 위 코드는 Store(a, i, v)의 j번째 값은, 만약 i == j라면 v이고, 아니라면 a[j]와 같다는 뜻임.

s.add(Store(a, i, v)[i] == v) # 바꾼 위치에서는 무조건 v로 바뀜을 확실히 해주는 것임.
```

즉,

- `i` 번째 인덱스는 `v` 로 변경됨,
- 다른 인덱스 `j` 는 원래 값 `a[j]` 유지됨.

배열의 "확장성(extensionality)"

- **확장성**이란, "두 배열이 모든 인덱스에서 같은 값을 가지면, 그 배열 전체도 같다"는 성질.

Z3는 이를 **명시적으로 강제**함.

```
s.add(Implies(ForAll(i, a[i] == b[i]), a == b))
```

즉, 배열 `a` 와 `b` 가 모든 `i` 에 대해 같으면, 두 배열 전체가 같다고 선언.

Skolem 함수 `Ext(a, b)` 도입

직접 모든 인덱스(i)를 비교할 수 없기 때문에, Z3는 "어디서 값이 다른지를 표시하는 Skolem 함수" `Ext(a, b)` 를 만들어 확장성 공리를 아래처럼 단순화함.

```
s.add(Implies(a[Ext(a, b)] == b[Ext(a, b)], a == b))
```

→ 즉, `Ext(a, b)` 가 "배열 a와 b의 차이를 나타내는 특정 인덱스" 역할을 함.

결과

이 변환을 통해 Z3는 배열 관련 식을 일반 **EUF 모델**로 처리할 수 있음.

즉,

- `Store` 연산은 "조건부 함수"로 바뀌고,
- `Ext` 는 "배열 간 차이를 추적하는 함수"로 작동하며, "두 배열이 모든 인덱스에서 동일하면 같다"는 논리적 일관성이 보장됨.

한 줄 요약

Z3는 배열 연산을 EUF 형태로 바꿔서 처리하며, Store는 조건식으로, 배열의 동일성은 확장성(Extensionality) 공리로 보장한다.

3.4 Bit Vectors

개념 요약

- **Bit Vector(비트 벡터)**는 고정된 길이의 2진수 형태 데이터를 다루는 Z3의 자료형이다.
- 즉, 정수를 비트 단위로 표현하여 **논리적 비트 연산(&, |, ^, <<, >> 등)** 으로 계산 가능.
- 비트 단위 최적화(Bit-fiddling)나 컴파일러 수준의 연산 검증에 자주 사용된다.

예시 1: 2의 거듭제곱 판별

```
def is_power_of_two(x):
    return And(x != 0, 0 == (x & (x - 1)))

x = BitVec('x', 4)
prove(is_power_of_two(x) == Or([x == 2**i for i in range(4)]))
```

설명:

- `(x & (x - 1)) == 0` 이면 `x` 는 **2의 거듭제곱**임을 의미.

  예: `8(1000b) & 7(0111b) = 0`
- `BitVec('x', 4)` : 4비트 비트벡터 선언
- `prove(...)` : 이 조건이 모든 경우에 대해 참임을 Z3가 증명함.

예시 2: 절댓값 계산 (비트 연산으로)

```
v = BitVec('v', 32)
mask = v >> 31
prove(If(v > 0, v, -v) == (v + mask) ^ mask)
```

설명:

- `v >> 31` : 부호 비트(sign bit) 추출 (양수면 0, 음수면 -1)
- `(v + mask) ^ mask` : **부호에 따라 절댓값을 만드는 비트 연산식**

- `v > 0` 이면 그대로
- `v < 0` 이면 반전 + 1 (즉, -v)

추가 설명

- `mask = v >> 31` 은 **산술적 시프트(arithmetic shift)**로, 부호 비트를 복사함.
- C언어나 하드웨어에서 정의되지 않는 연산(예: `v` overflow)도 Z3에서는 **논리적으로 완전(total)**하게 정의되어 있음.

한 줄 요약

Z3의 BitVec은 정수를 비트 단위로 표현해 논리적 비트 연산을 수행할 수 있게 하며, **이를 통해 2의 거듭제곱 판별, 절댓값 계산 등 저수준 연산 최적화를 논리적으로 검증할 수 있다.**

3.4.1 Solving Bit-vectors

핵심 개념

- Z3는 **bit-blasting** 기법을 사용해 비트 벡터 문제를 해결한다.
- **bit-blasting**: 비트 벡터의 각 비트를 **논리 변수(Propositional Variable)**로 변환하여, **명제 논리 (Propositional Logic)**로 푸는 방식.

예시: 비트 덧셈(Bit-vector addition)

- `v + w` (두 비트 벡터의 합)은 각 비트에 대해 **리플 캐리 가산기(ripple-carry adder)** 형태로 표현된다.
- 덧셈은 다음과 같은 관계로 표현됨:

$$out_i \leftrightarrow xor(x_i, y_i, c_i)$$
$$c_{i+1} \leftrightarrow (x_i \wedge y_i) \vee (x_i \wedge c_i) \vee (y_i \wedge c_i)$$
$$c_0 = 0$$

- 즉, 각 비트의 **합(out_i)**과 **올림(c_i)**을 논리식으로 정의하여, 전체 덧셈을 **논리절(clause)**로 변환해 처리한다.

요약

Z3는 비트 벡터 덧셈, 뺄셈 등 연산을 각 비트를 명제 논리식으로 변환(bit-blasting)하여 해결하며, 내부적으로 리플 캐리 구조를 사용해 carry(올림)을 계산한다.

3.4.2 Floating Point Arithmetic

개념

- 부동소수점(floating point) 수는 IEEE 부동소수점 표준(IEEE 754)에 따라 해석되는 **비트벡터(bit-vector)**로 표현된다.
- 즉, **비트 벡터 + 지수(exponent) + 가수(significand)** 구조를 사용.

```
x = FP('x', FPSort(3, 4))
print(10 + x)
```

이 선언은 **지수부(exponent)** 가 3비트, **가수부(significand)** 가 4비트인 부동소수점 수 `x` 를 정의한다.
`10 + x` 를 계산하면 결과는

$$1.25 \times (2^{**3}) + x$$

로 표현된다.

여기서 정수 10은 부동소수점 수로서 **지수(exponent) 3** (비트벡터 값 `011`),
가수(significand) `1010` 으로 표현된다.

즉, Z3는 부동소수점 연산을 비트벡터 기반으로 정확하게 모델링한다.

핵심 요약

Z3는 부동소수점 수를 비트 벡터 형태로 표현하며, IEEE 부동소수점 표준에 따라 지수와 가수를 구분하여 연산을 수행한다.

3.5 Algebraic Datatypes (대수적 데이터타입)

1차 대수적 자료형(first-order algebraic datatypes) 이론은 **유한 트리(finite trees)**의 이론을 포착(capture)한다.

이 이론은 다음과 같은 성질로 특징지어진다.

- **모든 트리는 유한하다** (occurs check).
- **모든 트리는 생성자(constructor)로부터 생성된다** (no junk).
- **두 트리가 같다는 것은, 동일한 방식으로 구성되었을 때에만 같다** (no confusion).

예시: 이진 트리 자료형 (Binary Tree Datatype)

```
Tree = Datatype('Tree')
Tree.declare('Empty')
Tree.declare('Node', ('left', Tree), ('data', Z), ('right', Tree))
Tree = Tree.create()
t = Const('t', Tree)
solve(t != Tree.Empty)
```

이 코드는 `Tree` 라는 새로운 자료형을 정의한다.

- `Empty` : 비어 있는 트리
- `Node(left, data, right)` : 왼쪽 서브트리, 데이터 값, 오른쪽 서브트리로 구성된 노드

Z3는 이 문제의 가능한 해를 다음과 같이 반환할 수 있다.

```
[t = Node(Empty, 0, Empty)]
```

즉, 왼쪽과 오른쪽이 비어 있고(`Empty`), 데이터 값이 0인 하나의 노드 트리를 의미한다.

또한, Z3를 사용하면 “트리가 자기 자신을 포함할 수 없다”는 사실도 증명할 수 있다.

```
prove(t != Tree.Node(t, 0, t))
```

즉, 어떤 트리 `t` 도 자기 자신을 부분으로 가질 수 없다.

요약

Z3의 대수적 자료형 이론은 유한 트리 구조를 안전하게 정의하고 조작하기 위한 이론이다. 트리는 생성자를 통해서만 만들어지고, 자기 참조나 무한 구조는 허용되지 않는다.

3.6 Sequences and Strings

핵심 개념

- Z3는 **문자열(String)**과 **시퀀스(Sequence)**를 다룰 수 있음.
- 문자열은 문자의 나열이고, 시퀀스는 **아무 타입(Int, Bool, String 등)**의 원소들을 순서대로 나열한 구조.
- 기본적으로 **연결(concatenation)**, **길이(length)**, **접두사(prefix)**, **접미사(suffix)** 등의 연산이 제공됨.

예제 1 : 문자열 연결 (Prefix + Suffix)

```
s, t, u = Strings('s t u')
prove(Implies(And(PrefixOf(s, t), SuffixOf(u, t),
              Length(t) == Length(s) + Length(u)),
                 t == Concat(s, u)))
```

만약 문자열 t의 길이가 접두사 `s` 와 접미사 `u` 의 길이 합과 같다면, `t` 는 `s` 와 `u` 의 연결(`Concat(s, u)`)이다.

예제 2 : 시퀀스(Sequence) 다루기

```
s, t = Consts('s t', SeqSort(IntSort()))
solve(Concat(s, Unit(IntVal(2))) == Concat(Unit(IntVal(1)), t))
prove(Concat(s, Unit(IntVal(2))) != Concat(Unit(IntVal(1)), s))
```

이건 **정수로 구성된 시퀀스(즉, Int 배열 비슷한 것)**를 예시로 보여주는 것임.

예제 2 코드 완전 해석

선언부

```
s, t = Consts('s t', SeqSort(IntSort()))
```

- `SeqSort(IntSort())` : “정수 시퀀스” 타입을 만듦
- `s` 와 `t` 는 각각 **정수 시퀀스 변수**

즉, `s = [ ... ]` , `t = [ ... ]` 같은 형태로 생각하면 돼요.

첫 번째 줄

```
solve(Concat(s, Unit(IntVal(2))) == Concat(Unit(IntVal(1)), t))
```

이건 이렇게 해석됨 : “s 뒤에 [2]를 붙인 시퀀스” == “앞에 [1]을 붙인 t 시퀀스”

즉,

```
s ++ [2] == [1] ++ t
```

을 만족하는 `s`, `t`를 찾아보라는 뜻이에요.

예를 들어,

`s = [1]`, `t = [2]`면 `[1] + [2] == [1] + [2]` -> 만족 (Z3가 찾을 수 있음)

두 번째 줄

```
prove(Concat(s, Unit(IntVal(2))) != Concat(Unit(IntVal(1)), s))
```

이번엔 반대로, “s 뒤에 [2]를 붙인 것”이 “앞에 [1]을 붙인 s”와는 절대 같을 수 없음을 증명하라.

즉,

```
[s] + [2] ≠ [1] + [s]
```

이건 자명하게 맞음,

- 왼쪽은 `[s₁, s₂, ..., 2]`
- 오른쪽은 `[1, s₁, s₂, ...]`

→ 첫 원소가 다름. 따라서 같을 수 없음. 그래서 Z3는 이 식을 참(True)으로 증명함.

문자열 솔버 설정

Z3에는 문자열 전용 솔버가 두 가지 있음 :

설정	설명
`smt.string_solver = "seq"`	Thai Trinh 버전
`smt.string_solver = "z3str3"`	Murphy Berzish 버전

전체 요약

Z3의 Sequences & Strings 이론은 문자열과 시퀀스를 다루는 논리적 모델로, 접두사/접미사, 연결(Concat), 길이(Length) 같은 연산을 포함한다.

시퀀스에서는 단일 원소(Unit)와 연결(Concat)을 통해 리스트처럼 연산할 수 있으며, 두 시퀀스가 같은 조건을 만족하는지 `solve`, `prove`로 확인할 수 있다.

3.7 Special Relations

관계(Relation)란?

Z3에서는 **x, y 두 원소 사이의 관계 R(x, y)**를 정의해서 “x ≤ y”, “x → y” 같은 논리적 관계를 표현할 수 있음.

예를 들어, “R이 부분 순서(partial order)”임을 정의하려면 세 가지 공리가 필요함.

```
s.add(ForAll([x], R(x, x))) # 반사성 reflexivity
s.add(ForAll([x, y], Implies(And(R(x, y), R(y, x)), x == y))) # 반대칭성 antisymmetry
s.add(ForAll([x, y, z], Implies(And(R(x, y), R(y, z)), R(x, z)))) # 추이성 transitivity
```

즉,

- 자기 자신과의 관계는 항상 성립해야 함 (xRx)
- xRy, yRx이면 x=y
- xRy, yRz이면 xRz

이게 **부분 순서의 정의임.**

문제점: 너무 느림

Z3는 이런 식으로 직접 공리를 넣으면, **모든 가능한 조합(x, y, z)** 에 대해 조건을 검사해야 해서 **너무 많은 인스턴스가 생김.**

예시 :

```
s.add(R(a1,a2), R(a2,a3), ..., R(a999,a1000))
```

이런 식으로 1,000개의 원소가 있다면, 추이성 때문에 (x,y,z) 세 쌍 조합을 다 검사해야 해서 **약 50만 개(=반백만)** 조건이 생겨버림

해결책: Z3의 "내장 관계" 사용하기

Z3는 이런 반복 계산을 피하려고, **부분 순서(Partial Order)**, **선형 순서(Linear Order)** 등을 **내장 기능으로 제공**함.

즉, 아래처럼 간단히 쓸 수 있음.

```
R = PartialOrder(A, 0)
```

이 한 줄이 위의 3개 공리 (반사성, 반대칭성, 추이성)를 **자동으로 내장한 R**을 만들어줌.

뒤에 있는 (A, 0)의 의미

`A` → 이 관계가 정의되는 **도메인(집합)**

`0` → "이름 구분용 인덱스(번호)"

예를 들어,

```
R1 = PartialOrder(A, 0)
R2 = PartialOrder(A, 1)
```

→ R1과 R2는 같은 `A` 위에 정의된 서로 다른 부분 순서 관계임.

즉, **같은 타입 위에 여러 관계를 정의할 수 있게 번호로 구분**하는 것임.

비슷한 내장 관계들

관계	의미	직접 쓰는 공리 대신
`PartialOrder(A, 0)`	부분 순서 (반사성, 반대칭성, 추이성)	R(x,x), xRy∧yRx→x=y, xRy∧yRz→xRz
`LinearOrder(A, 0)`	선형 순서 (모든 원소 쌍 비교 가능)	위 + xRy ∨ yRx
`TreeOrder(A, 0)`	트리 순서	위 + (xRy∧yRz) → (xRz ∨ R(y,z) ∨ R(z,y))
`PiecewiseLinearOrder(A, 0)`	구간별 순서	더 복잡한 조합 포함

즉, 위의 `ForAll(...)` 공리들을 다 직접 쓸 필요 없이, `R = LinearOrder(A, 0)` 처럼 한 줄로 선언하면 끝입니다.

핵심 요약

Z3는 관계(R)가 “부분 순서”, “선형 순서”, “트리 순서” 등의 성질을 자동으로 만족하도록 내장된 함수 (PartialOrder, LinearOrder, TreeOrder 등)를 제공한다.

이 방법은 모든 (x, y, z) 조합을 일일이 확인하는 대신, **Z3 내부의 그래프 기반 추론(graph reachability)** 으로 빠르게 처리한다.

3.8 Transitive Closure (추이 폐쇄)

기본 개념

추이적(transitive) : A → B, B → C 이면 A → C 이다 라는 관계의 성질

이때, 추이 폐쇄(transitive closure)는 “관계 R을 여러 번 이어붙였을 때, 연결될 수 있는 모든 쌍”을 포함한 관계

즉, “R을 1번, 2번, 3번, ... 여러 번 적용해서 도달 가능한 모든 관계를 한 번에 표현한 것”임.

예시로 이해해보기

관계	의미
R(a, b)	a → b
R(b, c)	b → c

이럴 때, 비록 **a → c**는 직접 R에 없지만, a → b → c 경로가 있으므로 **R의 추이 폐쇄에는 (a, c)** 가 포함됨.

즉, R^+ (또는 TC_R) = {(a,b), (b,c), (a,c)} → 이게 바로 R의 추이 폐쇄.

코드 설명

```
R = Function('R', A, A, B)       # 이항 관계 R(x, y)
TC_R = TransitiveClosure(R)      # R의 추이 폐쇄를 나타내는 관계 TC_R
s = Solver()
a, b, c = Consts('a b c', A)

s.add(R(a, b))    # a → b
s.add(R(b, c))    # b → c
s.add(Not(TC_R(a, c)))  # 그런데 a→c는 없다고 가정

print(s.check())  # UNSAT (즉, 모순 발생)
```

- a→b, b→c가 주어졌을 때 “a→c가 성립하지 않는다”라고 하면 모순(unsat)이 발생함
- 왜냐면 **추이성 때문에 a→c가 반드시 따라와야 하기 때문**