2. Logical Interfaces to Z3

Z3의 논리식 기초 이해

Z3란?

- Z3 : 논리식을 입력받아 그 식이 참이 될 수 있는지(=만족 가능한지) 계산해주는 논리 퍼즐 해석기
- 조건들을 동시에 만족할 수 있는 값이 존재하는지를 찾아줌.

개념 요약

개념	설명
논리식(formula)	참(True)/거짓(False)으로 판별할 수 있는 문장
원자 변수(atomic variable)	논리식의 기본 단위 (ex. `Tie`, `Shirt`)
논리 연산자(logical connective)	`And`, `Or`, `Not`, `Implies` 같은 연결어
Solver	조건들을 입력받아, 그것들이 동시에 가능한지 계산하는 Z3의 엔진
모델(model)	조건을 만족시키는 변수들의 값 조합

코드 예제

```
from z3 import *

# 불리언 변수 2개 생성
Tie, Shirt = Bools('Tie Shirt')

# Z3의 Solver 생성
s = Solver()

# 조건(제약식) 추가
s.add(Or(Tie, Shirt))          # Tie 또는 Shirt 중 하나는 참
s.add(Or(Not(Tie), Shirt))     # Tie가 아니면 Shirt는 참
s.add(Or(Not(Tie), Not(Shirt)))  # Tie가 아니면 Shirt는 거짓

# 식이 만족 가능한지 검사
print(s.check())   # → sat (satisfiable)
print(s.model())   # → 가능한 모델 출력
```

실행 결과

```
sat
[Tie = False, Shirt = True]
```

결과	의미
sat	식이 “만족 가능(satisfiable)”하다는 뜻
Tie = False, Shirt = True	이 조합일 때 모든 조건이 참이 됨

즉, "넥타이는 안 맸지만 셔츠는 입었다" → 세 문장을 모두 만족시키는 상황

`solve()` 함수

`solve()`는 다음 과정을 한 줄로 수행하는 **단축 함수**임.

```
Solver() → add() → check() → model()
```

예를 들어 다음 코드는 위 코드와 같은 일을 합니다:

```
solve(Or(Tie, Shirt),
      Or(Not(Tie), Shirt),
      Or(Not(Tie), Not(Shirt)))
```

Z3의 명제 논리 확장 : 배열과 산술

핵심 요약

명제 논리(Propositional Logic)는 Z3가 다루는 공식 중 가장 기초적인 부분임.

하지만 Z3는 그보다 더 강력하게 **산술(arithmetic)**, **배열(array)**, **함수(function)** 등의 이론이 섞인 논리식도 처리할 수 있습니다.

코드 예제

```
from z3 import *

Z = IntSort()                  # 정수형 정렬(Int sort)
f = Function('f', Z, Z)          # 정수 -> 정수 함수 f
x, y, z = Ints('x y z')          # 정수 변수 3개 선언
A = Array('A', Z, Z)             # 정수 인덱스, 정수 값 배열 A

fml = Implies(x + 2 == y, f(Store(A, x, 3)[y - 2]) == f(y - x + 1))
solve(Not(fml))
```

결과

```
unsat
```

즉, 이 식의 부정(Not(fml))이 충족 불가능(unsatisfiable) 하다는 것은 원래의 식 `fml`이 **항상 참(valid)** 이라는 뜻임.

식의 의미 해석

구성 요소	설명
Store(A, x, 3)	배열 `A`의 인덱스 `x` 위치에 값 `3`을 저장
[y - 2]	배열의 `y - 2`번째 인덱스의 값을 참조
Implies(조건, 결과)	"만약 조건이 참이라면, 결과도 참이어야 한다"는 의미

논리적 풀이 요약

1. 조건 `x + 2 == y`가 참이면 오른쪽 식 `f(Store(A, x, 3)[y - 2]) == f(y - x + 1)`을 확인
2. `y`를 `x + 2`로 대입하면 `f(Store(A, x, 3)[x]) == f(3)` 이 됨
3. 배열 `A`의 인덱스 `x`에 `3`을 저장했으니, `Store(A, x, 3)[x]`의 값은 `3`
4. 결국 식은 `f(3) == f(3)` → 항상 참

결론

fml은 어떤 x, y, z, 배열 A, 함수 f를 넣어도 항상 참이다. 즉, **항상 성립하는 논리식 (valid formula)**이다.

Z3와 SMT-LIB2 표준

핵심 요약

Z3는 SMT-LIB2 표준 형식을 따름. SMT-LIB2는 **1차 다정렬 논리(first-order multi-sorted logic)**를 기반으로, 다양한 **이론(theory)** - 정수, 실수, 배열, 비트벡터 등 - 을 조합하여 논리식을 표현하는 **표준 언어**임.

SMT-LIB2란?

- **SMT-LIB2 (Satisfiability Modulo Theories Library v2.6)**는 SMT(이론 기반 만족성 검사기)들이 공통으로 이해할 수 있는 형식을 정의한 **텍스트 기반 표준**임.
- 다양한 논리 조합을 지원하며, 각 조합은 별도의 이름으로 구분됨.

예 : SMT-LIB2의 QF_LIA (Quantifier-Free Linear Integer Arithmetic)

이는 정수형 변수에 대해 양/음수 연산과 비교연산을 허용하는 정수 산술의 일부분임.

항목	설명
QF	Quantifier-Free (즉, `∀`, `∃` 같은 양화사 없음)
LIA	Linear Integer Arithmetic (선형 정수 산술)
의미	정수 변수에 대해 `+`, `-`, `*`, `/`, `<`, `>`, `<=`, `>=` 등의 연산 가능

SMT-LIB 예시 코드

```
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (> (+ (mod x 4) (* 3 (div y 2))) (- x y)))
(check-sat)
```

- 의미 : x와 y가 정수일 때, (x % 4) + 3*(y/2) > x - y 인지 검사

Python(Z3 Py) 버전

```
from z3 import *
x, y = Ints('x y')
solve((x % 4) + 3 * (y / 2) > x - y)
```

- Z3의 `solve()` 함수가 자동으로 SMT-LIB2 형식으로 내부 변환하여 검사

Solver 상태를 SMT-LIB2 형식으로 출력하기

```
from z3 import *
x, y = Ints('x y')
s = Solver()
s.add((x % 4) + 3 * (y / 2) > x - y)
print(s.sexpr())
```

출력 결과

```
(declare-fun y () Int)
(declare-fun x () Int)
(assert (> (+ (mod x 4) (* 3 (div y 2))) (- x y)))
```

- 즉, `sexpr()`는 현재 Solver 내부 상태를 **SMT-LIB2 표준 표현으로 직렬화(serialize)** 해줌.

2.1 Sorts (정렬)

핵심 요약

- Sort(정렬)은 Z3에서 사용하는 데이터 타입(type) 개념임.
- SMT-LIB2 공식은 몇 가지 **기본 정렬**과 **사용자 정의 정렬**을 지원함.

기본 정렬 종류

정렬 이름	설명
Bool	참(True)/거짓(False)을 표현
Int	정수형
Real	실수형
BitVec(n)	`n`비트 크기의 비트 벡터 (예: 8비트 정수)
Array(Index, Elem)	인덱스 정렬과 원소 정렬을 가지는 배열
String / Seq(S)	문자열 또는 시퀀스 (S는 원소의 정렬)

사용자 정의 정렬 (DeclareSort)

Z3에서는 새로운 정렬을 직접 정의할 수도 있습니다.

```
S = DeclareSort('S')   # 새로운 정렬 S 정의
s = Const('s', S)      # S 타입의 상수 s 선언
solve(ForAll(s, s != s))
```

실행 결과

```
unsat
```

- 즉, "모든 s에 대해 s ≠ s" 라는 조건은 모순이므로 충족 불가능(unsatisfiable)임. (자기 자신과 같지 않은 원소는 존재할 수 없기 때문)

2.2 Signatures (서명 / 기호 정의)

핵심 요약

- Signature(서명)은 Z3에서 사용하는 기호(Symbol)들의 의미(타입, 매개변수, 반환값)를 정의한 것임.
- 즉, 어떤 이름이 **상수인지, 함수인지, 어떤 타입을 다루는지**를 정하는 단계임.

해석된 vs 자유로운 기호

구분	설명	예시
해석된 기호 (Interpreted symbol)	이미 의미가 정의되어 있음	`0`, `28`, `+`, `-`, `>`
자유 기호 (Free symbol)	사용자가 직접 의미를 부여해야 함	`x`, `y`, `f`, `A` 등

상수(Constant)

- 상수는 인자가 없는 **nullary 함수**로 취급됨.
- 즉, 함수지만 입력이 없고 고정된 값을 반환하는 형태입니다.
- 예:

```
x = Int('x')  # 자유 상수 (자유 변수)
```

함수(Function)

- 여러 인자를 받을 수 있는 **사용자 정의 함수**를 만들 수 있음.
- 함수의 입력과 출력 정렬(Sort)을 지정해야 함.
- 예:

```
Z = IntSort()
f = Function('f', Z, Z)   # 정수 -> 정수 함수
```

  - 이 함수는 "정수를 입력받아 정수를 반환"하는 함수 선언을 의미함.

- 불리언 함수도 동일하게 정의할 수 있음.

```
Z = IntSort()
B = BoolSort()
p = Function('p', Z, B)   # 정수 -> 불리언 함수
```

2.3. Terms and Formulas (항과 공식)

핵심 요약

- Z3에서 공식(formula)은 항상 불리언(Boolean) 값을 결과로 가지는 식임.
- 하지만 Boolean이 아닌 식(정수, 배열 등)을 포함할 수도 있으며, **정렬(sorts)**만 올바르게 맞춰주면 됩니다.

예제 코드

```
from z3 import *

B = BoolSort()
f = Function('f', B, IntSort())  # Bool → Int 함수
g = Function('g', IntSort(), B)  # Int → Bool 함수
a = Bool('a')
solve(g(1 + f(a)))
```

결과 모델

```
[a = False, f = [else → 0], g = [else → True]]
```

기호	의미
`a = False`	불리언 변수 `a`의 값
`f`	모든 입력을 0으로 매핑하는 함수
`g`	모든 입력을 True로 매핑하는 함수

- 즉, 이 식은 "`a`가 거짓이고, `f`는 항상 0을, `g`는 항상 True를 반환하는 경우"에 참이 됨.

Z3의 기본 논리 연산자

연산자	의미
`And`	논리곱 (∧)
`Or`	논리합 (∨)
`Not`	부정 (¬)
`Implies`	함의 (→)
`Xor`	배타적 논리합
`a == b` (둘 다 Bool)	쌍방향 함의 (↔)

식 구조 탐색 유틸리티

Z3는 식의 내부 구조를 살펴볼 수 있는 여러 메서드를 제공합니다.

```
x = Int('x')
y = Int('y')
n = x + y >= 3

print("num args:", n.num_args())  # 인자의 개수
print("children:", n.children())  # 하위 노드(자식 식)
print("1st child:", n.arg(0))     # 첫 번째 인자
print("2nd child:", n.arg(1))     # 두 번째 인자
print("operator:", n.decl())      # 연산자 선언
print("op name:", n.decl().name()) # 연산자 이름
```

- 위 식 `x + y >= 3`의 구조를 Z3가 내부적으로 어떻게 표현하는지 확인할 수 있습니다.

2.4. Quantifiers and Lambda Binding (양화사와 람다 바인딩)

핵심 요약

- Z3는 양화사(Quantifier)와 람다(Lambda) 개념을 지원함.
- 양화사는 변수를 논리식 범위 내에서 "묶어주는" 역할을 하고, 람다는 특정 조건에 따라 함수를 동적으로 정의할 수 있게 해줌.

양화사 (Quantifiers)

종류	기호	의미
전칭 양화사	∀ (`ForAll`)	"모든 x에 대해 ..."
존재 양화사	∃ (`Exists`)	"어떤 x가 존재해서 ..."

양화사는 **변수의 범위(scope)**를 공식 내에서 제한함.

예제 1 : 해가 없는 경우

```
from z3 import *

x, y = Ints('x y')
solve([y == x + 1, ForAll([y], Implies(y <= 0, x < y))])
```

- 의미:

  "모든 `y ≤ 0`에 대해 `x < y`가 참이 되어야 한다"

  하지만 어떤 `x`를 선택하더라도 `y`가 0 이하일 때 `x < y`를 만족시킬 수 없으므로 **해가 없음** (`unsat`).

- 참고:

  양화사 안의 `y`는 외부의 `y`와 **다른 변수**로 간주됨.

  `y == x + 1`은 단순한 대입이 아니라 **논리 제약**임.

💡 **이게 무슨 소리지 ? 성립하는 것 같은데 ... ?**

핵심 포인트 1 : 양화사(ForAll)는 새로운 변수 스코프를 만든다
여기서 중요한 건, `ForAll([y], ...)` 안의 `y` **는 외부의 y와 완전히 다른 변수임.**
즉,

- 첫 번째 `y == x + 1`의 `y`는 "외부 y"
- 두 번째 `ForAll([y], ...)` 안의 `y`는 "내부 y"

이 둘은 이름이 같을 뿐, **다른 존재**임.
Z3 입장에서는 "두 개의 서로 다른 y 변수가 존재함"을 의미함.

핵심 포인트 2 : 논리적으로 가능한가?
`ForAll([y], Implies(y <= 0, x < y))` 의 뜻은 "모든 y가 0 이하일 때, x는 그 y보다 작아야 한다."

- `x < 0` (y = 0일 때)
- `x < -1` (y = -1일 때)
- `x < -2` (y = -2일 때)
- ...
- **모든 음수보다 작아야 함**

결국 "x는 모든 음수보다 작아야 한다" → 즉, x는 -∞보다 작아야 한다
하지만 그런 x는 존재하지 않음. 그래서 **unsat(충족 불가능)**이 됨.

핵심 포인트 3 — 왜 헷갈리냐면
우리 눈에는 `y == x + 1` 때문에 y가 "x보다 1 크다"니까 "당연히 x < y 아닌가?"라고 생각하게 되는데,
`ForAll([y], ...)` 안에서 쓰이는 y는 **그 y가 아니기 때문에** 그 관계는 적용되지 않음.
즉, `y == x + 1`은 **외부 y**에 대한 제약이고, `ForAll([y], ...)` 은 **다른 y 전체에 대해 조건을 요구**하는 것임.

---

4. 예제 2 : 해가 존재하는 경우

```
solve([y == x + 1, ForAll([y], Implies(y <= 0, x > y))])
```

결과 :

```
[x = 1, y = 2]
```

즉, `x = 1`, `y = 2`이면 모든 `y ≤ 0`에서 `x > y`가 참이 되므로 **해가 존재**함.

람다 바인딩 (Lambda Binding)

람다(Lambda)는 **함수형 표현을 동적으로 정의**할 때 사용함.

배열이나 수학적 함수를 선언적(논리적으로) 정의할 수 있음.

```
m, m1 = Array('m', IntSort(), IntSort()), Array('m1', IntSort(), IntSort())

def memset(lo, hi, y, m):
    return Lambda([x], If(And(lo <= x, x <= hi), y, Select(m, x)))
```

```
solve([m1 == memset(1, 700, z, m), Select(m1, 6) != z])
```

- 의미 :
  - `memset(1, 700, z, m)` → 배열 `m`의 인덱스 1~700까지의 구간을 z로 채움.
  - `Lambda([x], ...)` → `x`마다 다른 값을 반환하는 함수 정의.

부분	의미
`Lambda([x], ...)`	"이런 규칙으로 새 배열 만들어줘" (`x`는 **배열의 각 칸을 나타내는 임시 변수**)
`If(lo <= x <= hi, y, Select(m, x))`	"범위 안이면 y, 아니면 m[x]"
`memset(1,700,z,m)`	"m 배열에서 1~700만 z로 덮은 새 배열"
`m1 == memset(1,700,z,m)`	"m1은 그 새 배열이야"
`Select(m1,6) != z`	"그런데 m1의 6번째 값은 z가 아니래" → 모순

결과적으로, `Select(m1, 6) != z`는 **충족 불가능(unsat)**이 됨.

왜냐하면 `6`이 1~700 사이이므로 `m1[6]`은 반드시 `z`여야 하기 때문임.

내부 동작 – 람다 리프팅과 변환

Z3는 내부적으로

- **Lambda lifting**
- **Reynolds' defunctionalization**

기법을 이용해 고차 함수를 1차 논리 형태로 변환해 처리함.

즉, 복잡한 람다 표현식도 논리적 공식으로 바꿔서 증명할 수 있음.

예제 3 : 람다 + 양화사 증명

```
Q = Array('Q', IntSort(), BoolSort())
prove(Implies(ForAll(Q, Implies(Select(Q, x), Select(Q, y))), x == y))
```

- 의미 : "모든 배열 Q에 대해, Q[x]가 참이면 Q[y]도 참이라면, x와 y는 같다."

단계	의미
1. Q는 Int → Bool 배열	각 인덱스에 True/False를 저장
2. ForAll(Q, Implies(Select(Q, x), Select(Q, y)))	모든 배열에서 Q[x]=True → Q[y]=True 가 참
3. 논리적 의미	Q[x]가 True인데 Q[y]가 False인 배열은 존재할 수 없음
4. 결론	그러려면 x와 y가 같아야 함
5. 결과	Z3가 "x == y는 항상 참"이라고 증명

- Z3는 이를 내부적으로 `Lambda(z, z == x)` 형태의 인스턴스를 생성하여 자동으로 증명함.
  - 즉, Z3는 내부적으로 "Q를 전부 탐색"하진 않고, "이런 Q가 존재할 수 있다면 모순이 생긴다"는 걸 **람다 표현식으로 간단히 모델링**해서 증명함.