

2. Logical Interfaces to Z3

2. Z3의 논리적 인터페이스

Z3는 단일 정렬(simple-sorted) 공식(formula)을 입력으로 받으며, 여기에는 미리 정의된 의미를 가진 심볼들이 포함될 수 있습니다. 이러한 의미는 이론(theory)에 의해 정의됩니다.

이 절에서는 Z3에 입력으로 사용될 수 있는 논리식의 개념을 소개합니다.

기본적으로, 명제 논리식(propositional formula)은 원자 변수(atomic variable)와 논리 연산자(logical connective)를 이용해 구성됩니다.

Z3가 인식할 수 있는 명제 논리식의 예시는 다음과 같습니다.

```
from z3 import *
Tie, Shirt = Bools('Tie Shirt') # 불리언(참/거짓) 변수 2개 생성
s = Solver()                  # 문제 해결기(Solver) 만들기

# 조건 3개 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

print(s.check()) # 조건이 동시에 가능한지 확인
print(s.model()) # 가능하다면, 어떤 값일 때 가능한지 보여줌
```

이 예제에서는 두 개의 불리언 변수 `Tie` 와 `Shirt` 를 정의하고, `Solver` 객체를 생성하여 세 개의 조건(assertion)을 추가합니다.

$$(Tie \vee Shirt) \wedge (\neg Tie \vee Shirt) \wedge (\neg Tie \vee \neg Shirt)$$

`s.check()` 를 호출하면 `sat` 라는 결과가 출력됩니다.

이는 주어진 식들이 만족 가능한(satisfiable) 조합을 가진다는 의미입니다.

예를 들어, `Tie` 가 거짓(false)이고 `Shirt` 가 참(true)인 경우가 해당됩니다.

이러한 만족 가능한 모델은 `s.model()` 을 통해 확인할 수 있습니다.

편의를 위해 Z3의 Python 인터페이스는 여러 단축 함수(shorthand function)를 제공합니다.

예를 들어 `solve` 함수는 Solver를 자동으로 생성하고, 제약을 추가한 뒤, 충족 가능성 검사를 하고, 가능한 경우 모델을 출력합니다.

명제 논리는 Z3가 처리하는 공식 중 가장 기본적이지만 작은 부분집합에 불과합니다.

Z3는 배열(array), 산술(arithmetic) 등 여러 이론에서 나온 심볼이 결합된 공식도 처리할 수 있습니다.

```
Z = IntSort()
f = Function('f', Z, Z)
x, y, z = Ints('x y z')
A = Array('A', Z, Z)
```

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

이 식은 항상 참(valid)입니다.

즉, 어떤 정수 x, y, z , 배열 A , 함수 f 에 대해서도 참이 성립합니다.

비록 z 는 식에 등장하지 않지만, Z3는 정수 변수를 선언해야 하므로 포함되어 있습니다.

`Store(A, x, 3)`은 배열 A 의 x 위치에 3 을 저장하고, $[y - 2]$ 는 그 배열의 특정 인덱스를 선택한다는 의미입니다.

논리적으로, 가정 $x + 2 = y$ 하에, 식의 오른쪽은 다음과 같이 단순화됩니다.

$$f(Store(A, x, 3)[x]) = f(3)$$

즉, 배열의 특정 인덱스에서 값을 3으로 저장하고, 다시 그 위치를 참조하면 3이 나온다는 의미입니다.

따라서 식의 부정 `Not(fml)`은 **충족 불가능(unsat)**하게 됩니다.

Z3는 일반적으로 **SMT-LIB2 표준** 형식의 공식을 따릅니다.

이 표준(현재 버전 2.6)은 **1차 다정렬 논리(first-order multi-sorted logic)**와 여러 이론 조합을 정의하는 텍스트 형식을 지정합니다.

예를 들어, SMT-LIB2의 **QF_LIA (Quantifier-Free Linear Integer Arithmetic)** 논리는 정수형 변수에 대해 양/음수 연산과 비교연산을 허용하는 정수 산술의 일부분입니다.

SMT-LIB 예시 :

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

Python 버전 :

```
solve((x % 4) + 3 * (y / 2) > x - y)
```

Z3의 `sexpr()`를 사용하면 현재 솔버 상태를 SMT-LIB2 형식으로 출력할 수도 있습니다.

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

출력 결과는 다음과 같습니다.

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

2.1. Sorts (정렬)

일반적으로, SMT-LIB2 공식은 **유한한 개수의 간단한 정렬(simple sorts)** 집합을 사용합니다.

이에는 기본 내장 정렬인 **Bool**이 포함되며, 지원되는 이론들마다 고유한 정렬을 정의할 수 있습니다.

예를 들어 다음과 같습니다:

- **Int, Real**
- **BitVec(n)** - 양의 정수 n 크기의 비트 벡터
- **Array(Index, Elem)** - 인덱스 정렬과 원소 정렬을 지정하는 배열
- **String, Seq(S)** - 문자열 및 시퀀스 정렬 (S 는 원소의 정렬)

또한 사용자가 **새로운 정렬**을 선언할 수도 있습니다.

단, 이러한 정렬의 **도메인은 절대 비어 있을 수 없습니다**.

예를 들어, 다음 공식은 **충족 불가능(unsatisfiable)** 합니다:

```
S = DeclareSort('S')
s = Const('s', S)
solve(ForAll(s, s != s))
```

즉, s 라는 새로운 정렬을 선언하고 s 라는 상수를 정의한 후, s 가 자기 자신과 같지 않다고 주장하는 것은 모순이기 때문입니다.

2.2. Signatures (서명, 기호 정의)

공식(formula)은 **해석된(interpreted)** 함수와 **자유로운(free)** 함수 및 상수를 함께 포함할 수 있습니다.

예를 들어, 정수 상수 $0, 28$ 은 **해석된 값**이며, 이전 예시의 x, y 는 **자유 상수**입니다.

상수(constant)는 인자가 없는 **nullary 함수**로 간주됩니다. 함수가 인자를 가지고 정의할 수도 있습니다. 예를 들어,

```
f = Function('f', Z, Z)
```

이는 하나의 정수 인자를 받아 정수를 반환하는 함수 선언을 만듭니다.

논리식에 사용되는 불리언 함수도 동일한 방식으로 정의할 수 있습니다.

2.3. Terms and Formulas (항과 공식)

Z3에서 공식으로 사용되는 식들은 **불리언 정렬(Boolean sort)**을 가져야 합니다.

하지만 Boolean과 비-Boolean 정렬의 식들을 섞어서 사용할 수도 있습니다.

단, 정렬이 일치해야 합니다.

예를 들어 다음 코드를 봅시다.

```
B = BoolSort()
f = Function('f', B, Z)
```

```
g = Function('g', Z, B)
a = Bool('a')
solve(g(1 + f(a)))
```

Z3는 다음과 같은 **모델(model)**을 생성할 수 있습니다.

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

즉, `a` 는 `False`로, `f` 는 모든 인자를 0으로, `g` 는 모든 인자를 `True`로 매핑합니다.

Z3에는 표준 논리 연산자들이 내장되어 있습니다.

- `And`, `Or`, `Not`, `Implies`, `Xor`
- 또한, `a == b` (`a`와 `b`가 Boolean일 때)는 **쌍방향 합의(bi-implication)**로 처리됩니다.

Z3는 식의 구조를 탐색할 수 있는 여러 **유틸리티 함수**를 제공합니다.

각 함수 적용에는 **함수 선언(declaration)**과 **인자(arguments)**가 있습니다.

예를 들어,

```
x = Int('x')
y = Int('y')
n = x + y >= 3
print("num args:", n.num_args())
print("children:", n.children())
print("1st child:", n.arg(0))
print("2nd child:", n.arg(1))
print("operator:", n.decl())
print("op name:", n.decl().name())
```

이 코드는 식 `x + y >= 3`의 구조를 탐색하여 각 하위 항목(children)과 연산자(operator)를 출력합니다.

2.4. Quantifiers and Lambda binding (양화사와 람다 바인딩)

Z3는 전칭(**∀, ForAll**)과 존재(**∃, Exists**) 양화사를 지원합니다.

이들은 특정 변수들을 공식의 범위 내에서 묶어줍니다.

예를 들어:

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

이 식은 **해(solution)**를 가지지 않습니다.

왜냐하면 어떤 `x` 를 선택하더라도, `y` 에 음수나 0 이하의 값이 존재하여 `x < y` 를 만족시킬 수 없기 때문입니다.

여기서 중요한 점은, **양화사 내부의 y**와 **외부의 y**는 서로 다른 존재입니다.

즉, `y == x + 1` 은 단순한 대입문이 아니라 **논리적 제약식**입니다.

반면, 다음 식은 해를 가집니다:

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

이 경우, $x = 1$, $y = 2$ 가 해(solution)가 됩니다.

Z3는 또한 **λ(람다) 바인딩**을 지원합니다.

이는 배열 및 함수형 표현을 다룰 때 유용합니다.

예를 들어,

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

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])
```

이 예제에서 `memset` 은 **1부터 700까지의 인덱스 구간**에 대해 값을 y 로 설정하고, 그 외의 구간에서는 기존 `m`의 값을 유지하는 배열을 반환합니다.

즉, `Lambda([x], ...)` 는 특정 x 값마다 다른 함수를 생성합니다.

Z3는 **람다 리프팅(Lambda lifting)**과 **Reynold의 defunctionalization** 기법을 사용하여 고차 함수를 1차 논리로 변환해 처리합니다.

예를 들어, 다음 명제는 Z3에서 증명(prove) 가능합니다.

```
Q = Array('Q', Z, B)
prove(Implies(ForAll(Q, Implies(Select(Q, x), Select(Q, y))), x == y))
```

이는 “모든 Q에 대해, $Q[x]$ 가 참이면 $Q[y]$ 도 참일 때, x 와 y 는 같다”는 의미입니다.

Z3는 내부적으로 `Lambda(z, z == x)` 형태의 인스턴스를 생성하여 이를 증명합니다.