# 4. Interfacing with Solvers

Z3의 **Solver(솔버)**는 수식(formula) 집합을 관리하며, **만족 가능성(satisfiability)**과 **스코프(scope)**를 제어한다.

한 스코프 내에서 추가된 식은 `pop()`을 통해 철회할 수 있다.

## 4.1 Incrementality (점증적 검사)

- 솔버는 **점진적 방식(incremental)**으로 식의 만족 가능성을 검사할 수 있다.
- 첫 번째 `check()` 호출은 **one-shot solver**로 실행되지만, 이후 호출부터는 자동으로 **incremental solver**로 전환된다.
- SMT Core를 기본으로 사용하며, 단순한 유한 영역(bit-vector, Boolean 등)에 대해서는 `QF_FD` 논리 기반의 특수 솔버를 사용할 수 있다.

## 4.2 Scopes (스코프)

- `push()`와 `pop()`을 통해 **로컬 스코프 생성 및 복원**이 가능하다.
- `push()`로 추가된 명제는 대응되는 `pop()` 호출 시 철회된다.
- 예제

```
p, q, r = Bools('p q r')
s = Solver()
s.add(Implies(p, q))
s.add(Not(q))
print(s.check())  # unsat

s.push()
s.add(p)
print(s.check())  # sat
s.pop()
print(s.check())  # unsat
```

  - 결과: `unsat → sat → unsat`
- 즉, 스코프 내 추가 명제는 일시적으로 적용된다.

## 4.3 Assumptions (가정)

- 스코프를 쓰지 않고, 특정 리터럴을 **가정(assume)** 한 채 만족 가능성 검사가 가능하다.
- `assert_and_track(expr, name)`을 사용하면 명제 `expr`에 라벨 `name`을 붙여 추적할 수 있다.
  - `assert_and_track(expr, name)` = `Implies(p, q)` & p = 참
- 예제

```
p, q = Bools('p q')
s = Solver()
s.add(Not(q))
s.assert_and_track(q, p)
print(s.check())  # unsat
```

- `q`와 `¬q`의 모순으로 `unsat` 발생
- 하지만 결과 라벨은 `p`로 표시됨 (즉, `p`가 모순의 원인으로 추적됨)

## 4.4 Cores (코어)

- **unsat core**: 모순을 일으킨 **가정(assumptions)**의 부분집합을 반환
- 즉, "왜 unsat이 발생했는가?"의 원인을 추적하는 기능
- 예제

```
p, q, r, v = Bools('p q r v')
s = Solver()
s.add(Not(q))
s.assert_and_track(q, p)
s.assert_and_track(r, v)
print(s.check())       # unsat
print(s.unsat_core())  # [p]
```

- `q`와 `¬q`의 모순 발생
- `unsat_core()`는 모순의 원인인 라벨 `[p]`만 반환

- 기본적으로 최소 코어(minimal core)는 반환되지 않지만, 다음 설정으로 최소화 가능:

```
def set_core_minimize(s):
    s.set("sat.core.minimize", "true")   # 비트벡터 이론용
    s.set("smt.core.minimize", "true")   # 일반 SMT용
```

## 4.5 Models (모델)

- `s.check()`가 `sat`이면 Z3는 변수 및 함수의 값을 포함한 **모델(model)**을 생성한다.
- `s.model()`을 통해 모델 객체에 접근할 수 있으며, 상수/함수 해석을 확인할 수 있다.
- 예제

```
f = Function('f', IntSort(), IntSort())
x, y = Ints('x y')
s.add(f(x) > y, f(f(y)) == y)
print(s.check())
print(s.model())
```

- 가능한 모델:

```
[y = 0, x = 2, f = [0 → 3, 3 → 0, else → 1]]
```

- 모델의 항목들은 각 상수나 함수 선언을 **해석값(interpretation)**에 매핑한다.
- 함수의 경우 `entry()`와 `else_value()`를 통해 동작 방식을 볼 수 있다.
  - 예제

```
m = s.model()
num_entries = m[f].num_entries()
for i in range(num_entries):
   print(m[f].entry(i))
print("else", m[f].else_value())
```

  - 출력:

```
[0, 3]
[3, 0]
else 1
```

- 모델 내 식을 직접 평가하려면 `eval()` 사용:

```
print(m.eval(x), m.eval(f(3)), m.eval(f(4)))
```

  - 출력: `2, 0, 1`

## 4.6 Other Methods

Z3의 솔버(Solver)는 단순히 `check()`로 sat/unsat 결과만 제공하는 것이 아니라,

추가적인 진단, 증명, 상태 복제, 병렬화, 로딩, 결과 추론 등 다양한 부가기능을 제공한다.

### 4.6.1. Statistics (통계)

- `print(s.statistics())` : 솔버가 내부적으로 수행한 연산 통계를 확인할 수 있다.
- 결정 절차(decision procedure)의 **카운터 값**을 보여주며, Z3가 탐색 과정에서 어떤 연산을 얼마나 수행했는지 분석할 때 유용하다.
- 즉, Z3의 **검색 특성과 성능 병목**을 파악할 때 사용하는 도구.

### 4.6.2. Proofs (증명)

- `produce-proofs` 옵션을 활성화하면 Z3는 **논리적 증명 객체(proof object)** 를 생성할 수 있다.

```
s.set("produce-proofs", True)
s.add(φ)
```

```
assert unsat == s.check()
print(s.proof())
```

- `s.proof()` 는 Z3가 "왜 unsat인지"를 논리적으로 증명한 결과를 보여준다.
- SMT Core의 증명은 **fine-grained(세밀)** 하게 표현되며, 반면 **양화사 제거(quantifier elimination)** 같은 복잡한 절차(QSAT)는 **하나의 큰 opaque step(불투명한 블록)** 으로 표시된다.
- 즉, "Z3가 논리적으로 unsat임을 증명하는 내부 추론 과정"을 보는 기능이다.

### 4.6.3. Retrieving Solver State (솔버 상태 조회)

- 현재 솔버에 저장된 명제(assertions)는 `s.assertions()` 로, 단위 리터럴은 `s.units()` , 비단위 리터럴은 `s.non_units()` 로 확인 가능하다.
- 이 상태는 SMT-LIB2 형식으로 출력할 수 있다.

  ```
  print(s.sexpr())  # SMT-LIB2 형식으로 상태 출력
  ```

  - 디버깅이나 외부 툴 연동 시 유용하다.

### 4.6.4. Cloning & Multithreading (솔버 복제 및 병렬 처리)

- `s.translate(ctx)` 를 사용하면 현재 솔버 상태를 **새 컨텍스트(Context)**로 복제할 수 있다.
- 이 기능은 **독립적인 상태의 솔버를 병렬로 실행**할 때 유용하다.
- 주의
  - 같은 Context에서 생성된 객체들은 **thread-safe하지 않음**.
  - 하지만 서로 다른 Context로 생성된 객체는 **병렬 연산이 안전**하다.

→ 즉, 병렬 탐색이나 다중 스레드 연산 시 Context 단위로 분리해야 한다.

### 4.6.5. Loading Formulas (수식 로딩)

- `s.from_file()` 또는 `s.from_string()` 을 통해 외부 파일이나 문자열에서 제약식을 로드한다.
- 기본 형식 : **SMT2 format**
- `.dimacs` 확장자의 경우: **DIMACS propositional format**으로 자동 인식된다.

→ 외부 논리식 파일을 바로 읽어 Z3에 주입할 때 사용.

### 4.6.6. Consequences (결과 추론)

- `consequences()` 는 특정 조건(assumptions)이 참일 때 **논리적으로 따라오는 결과(consequences)** 를 찾아준다.
- 예제

```
a, b, c, d = Bools('a b c d')
s = Solver()
s.add(Implies(a, b), Implies(c, d))  # 배경식
print(s.consequences([a, c], [b, c, d]))
```

- 출력

```
(sat, [Implies(c, d), Implies(a, b), Implies(c, d)])
```

- 해석
  - `a`와 `c`가 참이라면, 논리적으로 `b`와 `d`도 참이 된다.
  - 즉, "이 조건들을 참으로 두면 어떤 값이 자동으로 고정되는가?"를 계산한다.
- 사용 예시
  - **제품 구성 시스템(Product Configuration)**
    - 특정 옵션을 고정하면 자동으로 결정되는 다른 옵션을 계산.
    - 예: "SUV 선택 시 4WD가 자동 활성화됨."
- **CDCL(Conflict Driven Clause Learning)** 기반 알고리즘으로 내부 구현되어, SAT 코어와 SMT 코어 각각에 별도 버전이 존재한다.

### 4.6.7. Cubes (큐브: 문제 분할 기법)

**Cube and Conquer** = "큰 문제를 논리적으로 나눠서 푸는 Divide & Conquer 방식"

- `cube()` 함수는 논리식을 **여러 하위 케이스(subproblem)**로 분리한다.
- 각 큐브는 "리터럴들의 결합"으로 표현되며, 탐색 공간을 작게 나누어 **병렬 처리** 또는 **탐색 효율 향상**에 사용된다.
- 코드

```
s = SolverFor("QF_FD")
s.add(F)
s.set("sat.restart.max", 100)

def cube_and_conquer(s):
    for cube in s.cube():
        if len(cube) == 0:
            return unknown
        if is_true(cube[0]):
            return sat
        is_sat = s.check(cube)
        if is_sat == unknown:
            s1 = s.translate(s.ctx)
            s1.add(cube)
            is_sat = cube_and_conquer(s1)
        if is_sat != unsat:
```

```
            return is_sat
    return unsat
```

- 각 `cube`는 분기된 하위 조건들의 집합을 의미하며, 모든 분기를 탐색하면서 sat/unsat를 병렬적으로 확인한다.

---

- 큐브 분기 제어

| 설정값 | 의미 | 설명 |
|---|---|---|
| **sat.lookahead.cube.cutoff** | 분할 한계 | 큐브 분할 깊이 제한 |
| **cube.depth** | 1 | 고정된 깊이만큼만 분기 |
| **cube.freevars** | 0.8 | 비단위 리터럴 비율에 따라 깊이 자동 조정 |
| **cube.fraction** | 0.4 | 적응형 큐브 분할 비율 |
| **cube.psat.clause_base** | 2 | 절(clause) 가중치에 사용되는 지수 기반 |

---

- 리터럴 선택 휴리스틱

  어떤 리터럴을 먼저 분기 기준으로 삼을지 결정하는 파라미터 :

  ```
  sat.lookahead.reward
  ```

  → “어떤 리터럴을 먼저 선택하면 탐색이 가장 효율적인가?”를 자동으로 조정한다.