5. Using Solvers

5. Using Solvers

이 장에서는 Z3의 다양한 알고리즘을 소개한다. 이 알고리즘들은 앞선 섹션에서 설명한 인터페이스를 기반으로 개발되었다.

5.1 Blocking evaluations (모델 차단)

모델(model)은 **solver의 상태를 정제(refine)** 하는 데 사용할 수 있음.

예를 들어, solver를 반복 호출하면서 **이미 얻은 해(model)**와 **동일한 상수값 할당**을 다시 탐색하지 않도록 차단할 수 있음.

```
def block_model(s):
   m = s.model()
   s.add(Or([f() != m[f] for f in m.decls() if f.arity() == 0]))
```

이 함수는 현재 모델 `m`을 얻은 뒤, 모든 0-인자 함수(즉, 상수)에 대해 "이전 모델과는 다른 값을 가져야 한다"는 제약을 추가함.

즉, 동일한 모델을 다시 탐색하지 않도록 **차단(Blocking)**함.

💡 **코드 분석**

- `m = s.model()`
  - solver `s`의 현재 모델(즉, 하나의 해)을 가져옴.
- `m.decls()`
  - 모델 안의 모든 변수(상수)를 가져옴.
  - 예를 들어, `x = 1, y = 2`였다면 `m.decls() = [x, y]`.
- `f.arity() == 0`
  - arity : 함수의 인자 개수
  - `f.arity() == 0` : 인자가 없는 함수 (상수)
- `[f() != m[f] for f in m.decls() if f.arity() == 0]`
  - 각 변수 f에 대해 "**지금 모델의 값과 다르게 만들어라**"는 조건을 만듬.
  - 예를 들어 현재 모델이 `x = 1, y = 2`라면 이 부분은 `[x() != 1, y() != 2]`가 됨.
- `Or([...])`
  - 이 리스트를 "논리적으로 OR(또는)"로 묶음.
  - 즉, **"x가 1이 아니거나, y가 2가 아니어야 한다"**
  - **적어도 하나는 기존 모델과 다른 값을 가져야 한다**는 의미임.

모델 전체가 아니라 **특정 변수(terms)**에 대해서만 모델을 차단할 수도 있음.

```
def block_model(s, terms):
    m = s.model()
    s.add(Or([t != m.eval(t, model_completion=True) for t in terms]))
```

💡 **코드 분석**

- `m.eval(t, model_completion=True)`
  - `m.eval(expr)` : **모델에서 표현식(expr)의 값을 계산**
  - `model_completion=True` : 모델에 없는 심볼이라도 **가능한 기본값을 채워서 계산**하게 해주는 옵션
    - 예를 들어 모델 `m`에 `x`만 있고 `y`는 정의 안 되어 있다면, `m.eval(y, model_completion=True)`은 자동으로 `y`에 대한 기본값(예: 0)을 가정해 평가할 수 있음.
- `[t != m.eval(t, model_completion=True) for t in terms]`
  - `terms` : 차단할 대상이 되는 식들의 리스트
    - 예를 들어 `terms = [x, y]`라면, 이 부분은 `[x != 1, y != 2]`처럼 됨 (현재 모델에서 x=1, y=2라고 가정하면).
  - 즉, 각 식 `t`가 **현재 모델에서의 값과 달라야 한다**는 조건식을 만듬.

이제 여러 개의 해(모델)를 순차적으로 탐색하는 루프를 만들 수 있음.

```
def all_smt(s, terms):
    while sat == s.check(): # 해가 존재하는 동안 반복
        print(s.model())
        block_model(s, terms)
```

하지만 이 방식의 단점은, solver 상태에 **새로운 lemma(보조정리)**들이 계속 누적되어, solver 상태가 점점 커진다는 것임.

Z3는 이러한 오버헤드를 피하기 위한 **내장된 모델 열거(enumeration)** 기능은 제공하지 않음.

하지만 **scope(push/pop)** 기능을 활용하여 메모리 효율적으로 유사한 효과를 낼 수 있음.

```
def all_smt(s, initial_terms):
    def block_term(s, m, t):
        s.add(t != m.eval(t, model_completion=True))
    def fix_term(s, m, t):
        s.add(t == m.eval(t, model_completion=True))
    def all_smt_rec(terms):
        if sat == s.check():
            m = s.model()
            yield m
            for i in range(len(terms)):
                s.push()
```

```
            block_term(s, m, terms[i])
            for j in range(i):
                fix_term(s, m, terms[j])
            yield from all_smt_rec(terms[i:])
            s.pop()
    yield from all_smt_rec(list(initial_terms))
```

💡 **코드 분석**

- `block_term(s, m, t)`
  - 현재 모델 `m`에서 `t`가 가진 값과 **다르게** 만들라는 제약을 추가.
  - 즉, **"이번 모델과 같은 `t` 값은 다시 금지"** 하는 한 줄.
- `fix_term(s, m, t)`
  - 현재 모델 `m`에서 `t`의 값을 **고정**.
  - 즉, **"지금 본 모델의 `t` 값과 동일하게 유지"**하라는 한 줄.
- `all_smt_rec(terms)`
  - 한 번 모델을 얻으면(`m`), 그 모델을 **씨앗(seed)** 으로 삼아 **분기**를 만듬.
  - 분기 방법 :
    - 앞에서부터 차례로 `terms[i]` 하나를 골라 **그 값만 "다르게"** 만들고(`block_term`),
    - 그보다 앞의 것들 `terms[0, ..., i-1]`은 **"방금 모델의 값으로 고정"** 함(`fix_term`).
    - 그러면 "앞쪽 값들은 동일, i번째는 다름"이라는 **서로 겹치지 않는 영역**이 생김.
  - 그 분기 안에서만 다시 같은 방식으로 재귀(`terms[i:]`) 돌려서 더 쪼갬.
  - 각 분기 시작/종료마다 `push/pop`으로 **제약을 롤백**하여, 누적 레마가 쌓이지 않고 메모리/성능을 지킴.

5.2 All-SMT with user propagators (사용자 전파자 기반 All-SMT)

user propagator 플러그인은 Boolean 및 Bit-vector 표현식에 대한 **할당 상태를 추적(track)** 할 수 있게 해줌.

이를 통해 solver는 **콜백(callback)**을 이용하여 결과를 전파하거나, **모순되는 할당 조합을 차단(block)** 할 수 있음.

여기서는 All-SAT 열거(enumeration)를 위해 **user propagator의 기본 동작**을 보여줌.

즉, 추적 중인 모든 표현식에 대한 **완전한 할당이 완료되면**, 그 조합을 **차단(conflict)**하여 새로운 조합을 강제로 탐색하게 됨.

예제 코드

```
class BlockTracked(UserPropagateBase):
    def __init__(self, s):
        UserPropagateBase.__init__(self, s)
        self.trail = []   # 현재까지의 변수 할당 기록
```

```
    self.lim = []     # push/pop을 위한 경계 스택

    # Boolean 또는 BitVec 값이 할당(fixed)될 때 호출되는 콜백 등록
    self.add_fixed(lambda x, v: self._fixed(x, v))

    # 모델이 완성(모든 변수 할당 완료)되었을 때 호출되는 콜백 등록
    self.add_final(lambda: self._final())

def push(self):
    self.lim += [len(self.trail)]

def pop(self, n):
    self.trail = self.trail[:self.lim[len(self.lim) - n]]
    self.lim = self.lim[:len(self.lim) - n]

def _fixed(self, x, v):
    # 변수 x에 값 v가 할당되면 trail에 추가
    self.trail += [(x, v)]

def _final(self):
    # 모든 변수의 할당이 완료되었을 때 실행
    print(self.trail)
    self.conflict([x for x, v in self.trail])  # 현재 조합을 차단
```

💡 **코드 분석**

- `__init__(self, s)`

```
def __init__(self, s):
    UserPropagateBase.__init__(self, s)
    self.trail = []   # 지금까지 할당된 (변수, 값) 목록
    self.lim = []     # push/pop 경계 스택
```

  - `s`는 solver 객체 (`SimpleSolver()` 같은 것)
  - `trail` : 지금까지 "할당된 변수와 그 값"을 순서대로 저장
  - `lim` : `push/pop` 호출 시 경계 지점을 기록 (나중에 되돌릴 때 사용)

- 콜백 등록 부분

```
self.add_fixed(lambda x, v: self._fixed(x, v))
self.add_final(lambda: self._final())
```

  Z3의 **user propagator 콜백 등록 메서드** 두 개:

  1. `add_fixed(callback)`
     - 어떤 Boolean이나 BitVec 변수가 특정 값으로 고정(fixed)될 때 호출됨.
     - 즉, solver가 "이 변수는 이제 True다" 또는 "이건 5로 확정됐다" 하는 순간 실행됨.
     - 여기서는 `_fixed(x, v)`를 호출하도록 설정함.
  2. `add_final(callback)`
     - 추적 중인 모든 변수가 값이 정해져서 **모델이 완성**되었을 때 실행.
     - `_final()`을 호출하도록 설정함.

- push / pop

```
def push(self):
    self.lim += [len(self.trail)]

def pop(self, n):
    self.trail = self.trail[:self.lim[len(self.lim) - n]]
    self.lim = self.lim[:len(self.lim) - n]
```

  이건 solver가 내부적으로 **branching (분기)** 할 때 사용되는 스택 관리 함수임.
  solver가 탐색 중에 `push()`를 호출하면 현재 trail 길이를 저장하고, 나중에 `pop(n)`을 호출하면 그 시점 이후의 trail을 잘라내서 되돌림.
  → 즉, "되돌리기(undo)"를 위한 스냅샷 기능.

- `_fixed(self, x, v)`

```
def _fixed(self, x, v):
    # 변수 x에 값 v가 할당되면 trail에 추가
```

```
        self.trail += [(x, v)]
```

solver가 어떤 변수를 확정시킬 때마다 이 함수가 불림.
`x`와 `v`(즉, 변수와 그 값)을 `trail` 리스트에 추가.

- `_final(self)`

```
def _final(self):
    print(self.trail)
    self.conflict([x for x, v in self.trail])
```

  - **모든 변수의 값이 결정되어** 모델이 완성되면 자동으로 호출됨.
  - `print(self.trail)` : 지금까지의 변수-값 조합을 출력.
  - `self.conflict([...])` :
    → Z3에게 "이 조합은 **충돌(conflict)** 상태이므로 허용하지 마라"라고 통보함.
  - 즉, 현재 모델이 완성된 순간에 "이 모델은 금지!"라고 표시해서 다음 `check()` 때는 **이와 다른 조합**을 강제로 찾게 만드는 것.

즉, block_model() 함수의 고급 버전임.
`block_model`은 외부에서 수동으로 모델을 막았고, `BlockTracked`는 solver 내부의 propagator 단계에서 자동으로 막음

동작 설명

이 코드는 `UserPropagateBase`를 상속받아,

Z3의 내부 **전파 메커니즘(propagation mechanism)** 을 사용자 정의 방식으로 확장한다.

- `add_fixed`: Boolean 변수가 특정 값으로 **고정(fixed)** 될 때 호출됨.
- `add_final`: 모든 추적 변수에 값이 할당되면 호출되어,
  `self.conflict()`로 현재 모델을 **차단(conflict)** 하여 다음 모델 탐색을 유도함.
- `push/pop`: solver의 스코프(stack)를 관리하여 되돌릴 수 있게 함.

예제 실행

```
s = SimpleSolver()
b = BlockTracked(s)

x, y, z, u = Bools('x y z u')
b.add(x)
b.add(y)
b.add(z)

s.add(Or(x, Not(y)), Or(z, u), Or(Not(z), x))
print(s.check())
```

예시에서는 4개의 Bool 변수를 선언하고, 이 중 3개(x, y, z)를 propagator가 추적한다.

solver는 가능한 모든 할당 조합을 시도하며, 이미 탐색된 조합은 `conflict()`를 통해 차단된다.

5.3 Maximizing Satisfying Assignments (만족할 수 있는 해를 최대화하기)

모델을 활용하는 또 다른 방법은, **'최적 모델(optimal model)'**의 개념을 도입하는 것임.

어떤 식 집합 `ps`가 있을 때, `mss`(maximal satisfying subset, 최대 만족 부분집합)는 solver 상태 `s`와 **모순되지 않으며**, 그 어떤 식도 더 추가할 수 없는(추가하면 unsat이 되는) **최대의 만족 가능한 부분집합**임.

알고리즘 개요

다음 코드는 **최대 만족 부분집합(MSS)** 탐색 절차를 구현한 것임.

```
def tt(s, f):
    return is_true(s.model().eval(f))

def get_mss(s, ps):
    if sat != s.check():
        return []
    mss = { q for q in ps if tt(s, q) }
    return get_mss(s, mss, ps)

def get_mss(s, mss, ps):
    ps = ps - mss
    backbones = set([])
    while len(ps) > 0:
        p = ps.pop()
        if sat == s.check(mss | backbones | {p}):
            # p를 추가해도 만족된다면 MSS에 추가
            mss = mss | {p} | {q for q in ps if tt(s, q)}
            ps = ps - mss
        else:
            # 추가 시 모순이라면 backbone(항상 참/거짓으로 고정된 식)에 기록
            backbones = backbones | {Not(p)}
    return mss
```

💡 **코드 분석**

전체 맥락

- `s`는 이미 어떤 전제(제약)들이 들어 있을 수 있는 Z3 `Solver`
- `ps`는 "추가로 넣어볼 후보 식들(assumptions/constraints)의 집합"
- 목표: 현재 `s`의 전제와 **모순 없이 함께 참일 수 있는** 식들을 최대한 많이 담은 집합 `mss`를 구하기

함수별 역할

- `tt(s, f)`

```
def tt(s, f):
    return is_true(s.model().eval(f))
```

  - 방금 `s.check()`가 `sat`였을 때 사용할 수 있음.
  - 현재 모델에서 식 `f`가 **참인지** 평가해 `True/False` 반환.
  - 빠르게 "지금 모델이 이미 만족시키는 식들"을 거둬담는 데 사용.

- 진입점 `get_mss(s, ps)`

```
def get_mss(s, ps):
    if sat != s.check():
        return []
    mss = { q for q in ps if tt(s, q) }
    return get_mss(s, mss, ps)
```

  - 먼저 현재 `s`가 **만족 가능**인지 확인. (아예 UNSAT이면 빈 집합 반환)
  - `mss`를 "지금 모델이 **이미** 참으로 만드는 `ps`의 원소들"로 초기화.
    - 즉, 검증 없이 공짜로 넣을 수 있는 것들부터 담는다.
  - 그리고 실질 절차(아래 오버로드된 `get_mss`)로 넘어가 **확장/검증**을 시작.

- 핵심 절차 `get_mss(s, mss, ps)`

```
def get_mss(s, mss, ps):
    ps = ps - mss
    backbones = set([])
    while len(ps) > 0:
        p = ps.pop()
        if sat == s.check(mss | backbones | {p}):
            # p를 추가해도 SAT이면, p를 mss에 넣고
            # 지금 모델에서 이미 참인 나머지도 우르르 같이 담는다(가속)
            mss = mss | {p} | {q for q in ps if tt(s, q)}
            ps = ps - mss
        else:
            # p를 더하면 UNSAT → p는 들어갈 수 없음.
            # 그 사실을 "backbone"으로 기록: Not(p)가 항상 필요함을 의미
```

```
            backbones = backbones | {Not(p)}
    return mss
```

- 처음에 `ps`에서 이미 담은 `mss`를 제외.
- `backbones`는 "항상 유지되어야 하는 리터럴들을 축적"하는 집합.
  - 여기서는 `p`가 못 들어가면 `Not(p)`를 backbone에 추가(= "p는 반드시 거짓이어야 한다"라는 사실을 기억).
- 루프:
  1. 어떤 후보 `p`를 하나 뽑아, `mss ∪ backbones ∪ {p}`가 **SAT인지** 본다.
  2. SAT라면:
     - `p`를 `mss`에 넣는다.
     - **가속 트릭**: 지금 모델에서 이미 참인 `ps`의 나머지 `q`들도 한꺼번에 더 담는다(`tt(s, q)`).
       - 이유: 같은 `check` 결과 모델을 이용해 공짜로 담을 수 있는 걸 최대한 빨리 담아 탐색을 줄이기 위함.
     - `ps`에서 이제 담긴 것들을 제거하고 계속.
  3. UNSAT라면:
     - `p`는 절대 함께 갈 수 없음 → `Not(p)`를 backbone에 추가("p는 항상 배제돼야 함").
     - `ps`는 그대로 두고 다음 후보로 진행.
- 모든 후보를 처리하면 **더 이상 추가 가능한 식이 없으므로** 현재 `mss`가 **최대 만족 부분집합**이 되어 반환.

작동 예(직관)
예를 들어

- 현재 제약: `s`는 비워두고,
- 후보들: `ps = { p, q, r }`,
- 실제로 동시에 가능한 건 `{p, r}` 뿐이라고 하자.

흐름(한 가지 가능 시나리오):

1. `s.check()` → SAT, 모델 M0에서 `tt(s, q) == False`, `tt(s, p) == True`, `tt(s, r) == True` 라고 가정.
   ⇒ 초기 `mss = {p, r}`.
2. `ps - mss = {q}` 남음, `backbones = ø`.
3. `p = q` 뽑아 `s.check({p, r} ∪ ø ∪ {q})` = `s.check({p, r, q})`가 UNSAT → `backbones |= {Not(q)}`.
4. 더 볼 것 없음 → `mss = {p, r}` 반환 (최대: 더 넣을 수 있는 후보가 더 이상 없음)

5.4 All Cores and Correction Sets (모든 코어와 보정 집합)

Marco 절차는 **모델(models)**과 **코어(cores)**를 결합하여, 모든 **unsatisfiable core(모순 코어)**와 모든 **최대 만족 부분집합(MSS, Maximal Satisfying Subsets)**을 나열(enumerate)함.

이는 주어진 식 집합 `ps`에 대해, solver `s`를 기준으로 각 부분집합의 상태를 탐색함.

이 알고리즘은 `map`이라는 별도의 solver를 유지하여, `ps`의 어떤 부분집합이 아직 **코어의 상위집합(superset)**이거나 **MSS의 하위집합(subset)**인지 판별되지 않았는지를 추적함.

코드 설명

```
def ff(s, p):
    return is_false(s.model().eval(p))

def marco(s, ps):
    map = Solver()
    set_core_minimize(s)
    while map.check() == sat:
        seed = {p for p in ps if not ff(map, p)}
        if s.check(seed) == sat:
            mss = get_mss(s, seed, ps)
            map.add(Or(ps - mss))
            yield "MSS", mss
        else:
            mus = s.unsat_core()
            map.add(Not(And(mus)))
            yield "MUS", mus
```

코드 분석
이 코드는 **Marco algorithm** - 즉, **모든 MSS(최대 만족 부분집합)**과 **모든 MUS(최소 모순 부분집합)**을 교대로 탐색하는 절차임.

- `ff(s, p)`

```
def ff(s, p):
    return is_false(s.model().eval(p))
```

  - `s.model().eval(p)` : 현재 모델에서 식 `p`의 값을 평가한다.
  - `is_false(...)` : 그 값이 거짓인지 확인.
  - 즉, ff(s, p)는 "현재 모델에서 p가 거짓인가?"를 반환.
- `marco(s, ps)` - 핵심 루프
  - 이 함수는 제약식 집합 `ps`를 바탕으로 MSS/MUS를 반복해서 찾아냄.

- `map = Solver()`
  - 보조 solver.
  - `map`은 어떤 조합이 이미 탐색됐는지 관리하는 "지도(map)" 역할.
  - `map`의 제약은 "이 영역은 이미 처리했으니 건너뛰자"는 정보를 담음.
- `set_core_minimize(s)`
  - solver `s`에게 **unsat core 최소화** 기능을 활성화하는 설정.
  - 즉, `s.unsat_core()`가 반환할 때 가능한 한 작은 코어(최소 모순 집합)를 반환하도록 함.
- `while map.check() == sat:`
  - `map`이 **아직 탐색할 새로운 영역(seed)** 을 찾을 수 있을 때 반복.
- `seed = {p for p in ps if not ff(map, p)}`
  - `map`의 현재 모델에서 거짓이 아닌(즉, 참인) 식들만 뽑아서 seed를 구성.
  - 이 seed는 "다음 탐색의 시작점(후보 부분집합)"이 된다.
  - 즉, "아직 처리되지 않은 조합 중 하나의 부분집합(seed)"을 map이 제시하고, 그 seed가 SAT인지 UNSAT인지에 따라 분기를 타는 구조.
- SAT 가지 → MSS 탐색

```
if s.check(seed) == sat:
    mss = get_mss(s, seed, ps)
    map.add(Or(ps - mss))
    yield "MSS", mss
```

  1. `s.check(seed)` → seed 조합이 **만족 가능한 경우**.
  2. 그럼 seed를 시작으로 **최대 만족 부분집합(MSS)**을 `get_mss`로 확장.
     - `get_mss`는 SAT일 때 가능한 최대 조합을 구함
  3. `map.add(Or(ps - mss))`

- 즉, “MSS에 속하지 않는 식들 중 적어도 하나는 포함돼야 한다”는 조건을 추가.
- 이건 “이 MSS보다 더 큰 조합은 이제 탐색 안 함”을 의미.

4. `yield "MSS", mss`
   - 결과를 반환(출력).
   - 예: `("MSS", {p1, p3, p4})`

- UNSAT 가지 → MUS 탐색

```
else:
    mus = s.unsat_core()
    map.add(Not(And(mus)))
    yield "MUS", mus
```

1. `s.check(seed)`가 `unsat`이라면, 이 seed 조합은 **모순(UNSAT)**을 일으킨다는 뜻.
2. `s.unsat_core()`로 그 **모순의 원인 최소 집합(MUS)**을 얻음.
3. `map.add(Not(And(mus)))`
   - 즉, “이 MUS 전체가 동시에 참이 되는 조합은 다시 시도하지 말라.”
   - `Not(And(...))` = MUS 전체를 다시는 함께 켜지 않게 막는 제약
4. `yield "MUS", mus`
   - 결과를 반환.

   ◦ 예: `("MUS", {p1, p2})`

- 루프 반복
  - MSS 또는 MUS를 찾을 때마다 `map`에 새로운 제약을 추가.
  - 그러면 `map.check()`가 점점 더 많은 영역을 “탐색 완료”로 간주하게 됨.
  - 모든 가능한 부분공간이 커버되면 `map.check()`가 `unsat`이 되어 루프 종료.

예시 흐름
제약식 집합 ps = {a, b, c} 라고 하자.
여기서 `a`와 `b`는 서로 모순이고, `c`는 독립적이라면 (a ∧ c, b ∧ c)

단계	seed	SAT 결과	탐색 결과	map에 추가된 제약	의미
1	`{a}`	✅ SAT	MSS = {a,c}	`Or(b)`	“a쪽 영역 커버됨 → 이제 b쪽만 탐색”
2	`{b}`	✅ SAT	MSS = {b,c}	`Or(a)`	“b쪽 영역 커버됨 → a,b 같이일 때만 남음”
3	`{a,b}`	❌ UNSAT	MUS = {a,b}	`Not(And(a,b))`	“a,b 둘이 동시에 불가능 → 차단”
4	—	—	탐색 종료	—	더 이상 새로운 seed 없음

즉, 이 절차는 **MSS(만족 가능한 최대 집합)**과 **MUS(모순 집합)**을 교대로 찾으며, 모든 가능한 조합을 효율적으로 탐색한다.

5.5 Bounded Model Checking (한정된 모델 검사)

Bounded Model Checking (BMC)은 **전이 시스템(transition system)**을 입력받아 특정 **목표 상태(goal)**가 도달 가능한지를 확인하는 절차이다.

기본 정의

전이 시스템은 다음 4요소로 구성된다:

$$\langle Init, Trans, Goal, \mathcal{V} \rangle$$

- **Init**: 초기 상태를 정의하는 술어(predicate)
- **Trans**: 상태 간 전이 관계 (transition relation)
- **Goal**: 도달하려는 목표 상태
- $\mathcal{V}$: 변수들의 집합

도달 가능한 상태 집합은 다음과 같이 귀납적으로 정의된다:

- $s \models Init$, 즉 `s`가 초기 상태이거나
- 어떤 이전 상태 s_0와 $Trans(s_0, s)$를 만족하는 값 v가 존재하면, `s`는 도달 가능한 상태이다.

Z3에서는 다음과 같이 `init`, `trans`, `goal`을 이용해 BMC를 구현할 수 있다.

```
index = 0
def fresh(s):
    global index
    index += 1
    return Const("!f%d" % index, s)

def zipp(xs, ys):
    return [p for p in zip(xs, ys)]

def bmc(init, trans, goal, fvs, xs, xns):
    s = Solver()
    s.add(init)
    count = 0
    while True:
        print("iteration ", count)
        count += 1
        p = fresh(BoolSort())
        s.add(Implies(p, goal))
        if sat == s.check(p):
            print(s.model())
            return
        s.add(trans)
        ys = [fresh(x.sort()) for x in xs]
```

```
        nfvs = [fresh(x.sort()) for x in fvs]
        trans = substitute(trans,
                           zipp(xns + xs + fvs, ys + xns + nfvs))
        goal = substitute(goal, zipp(xs, xns))
        xs, xns, fvs = xns, ys, nfvs
```

💡 **코드 분석**

- `fresh(s)`

```
index = 0
def fresh(s):
    global index
    index += 1
    return Const("!f%d" % index, s)
```

  - 매 호출마다 고유한 이름의 상수 `!f1, !f2, ...` 를 **주어진 sort** `s` 로 만듦.
  - 상태 변수/입력 변수의 **새 타임스텝(프레임)** 용 복사본을 만들 때 사용.

- `zipp(xs, ys)`

```
def zipp(xs, ys):
    return [p for p in zip(xs, ys)]
```

  - `substitute` 에 넣을 **대응쌍 리스트**(원래식 → 치환식)를 만들기 쉽게 도와줌.

- BMC 본체

```
def bmc(init, trans, goal, fvs, xs, xns):
```

  - `init` : 초기 상태 술어. 보통 `xs` 만 등장. (예: `x0 == 0`)
  - `trans` : 전이 관계 (T(x, x')). 현재 `xs` 와 다음 상태 `xns` (= x')에 걸리는 식.
  - `goal` : 목표 상태 술어. 보통 현재 상태 `xs` 에 대해 정의.
  - `fvs` : 각 프레임마다 새로 생기는 **자유(입력) 변수들**(nondet inputs). 없으면 `[]` .
  - `xs` : "현재 프레임"의 상태 변수 목록 (예: `[x0]`)
  - `xns` : "다음 프레임(= 현재의 x')"의 상태 변수 목록 (예: `[x1]`)

- 루프 구성

```
s = Solver()
s.add(init)             # 프레임 0 초기조건 고정
count = 0
while True:
    print("iteration ", count)
    count += 1
    p = fresh(BoolSort()) # 활성화 리터럴(가드) p_k
    s.add(Implies(p, goal))
    if sat == s.check(p):
        print(s.model())  # 프레임 k에서 goal 달성 경로(모델) 출력
        return
```

  - **프레임 k에서 goal 검증** :
    `p` 는 "이번 프레임의 goal을 **시험**하는 스위치" 같은 것.

`Implies(p, goal)`을 추가하고 `s.check(p)`로 "p를 참으로 가정"하면, "현재까지 언롤된 경로 위에서 `goal`이 참인가?"를 묻는 것과 같음.

- `sat`이면: 지금까지 쌓인 전이들로 **goal 도달 경로가 존재** → 모델 출력 후 종료.
- `unsat`이면: 아직은 못 감.

> 왜 goal을 바로 s.add(goal) 하지 않고 Implies(p, goal) + s.check(p)로 하냐? → `goal`을 **이 프레임에서만** 시험하고 싶기 때문.
> 다음 프레임에서도 새 `p`로 다시 시험해야 하니까, `goal`을 영구 제약으로 못 박으면 안 됨.

- 프레임 확장(전이 언롤링)

```
s.add(trans)  # 지금 프레임의 T(xs, xns) 추가

ys   = [fresh(x.sort()) for x in xs]   # 다음 프레임의 x'가 그 다음 프레임에선 "현재"가 됨
nfvs = [fresh(x.sort()) for x in fvs]  # 입력(자유) 변수도 프레임마다 새로 만듦
trans = substitute(trans,
                   zipp(xns + xs + fvs, ys + xns + nfvs))
goal  = substitute(goal, zipp(xs, xns))
xs, xns, fvs = xns, ys, nfvs
```

프레임을 k → k+1로 **한 칸 밀기(시프팅)** 하는 부분 :

1. `s.add(trans)`
   - 지금까지의 "현재 상태 `xs` → 다음 상태 `xns`" 전이가 실제로 **한 단계 적용**됨.
   - 즉, 프레임 경로가 **한 스텝 길어졌음**.
2. `ys = fresh(...)`
   - 새 프레임의 "다음 상태" 자리를 위한 신선한 변수들 생성.
   - 이름상으로 보면: `xs`(현재), `xns`(다음), `ys`(다다음의 자리).
3. `nfvs`
   - 자유변수(입력)도 프레임마다 새 심볼을 씀 (시간에 따라 값이 달라질 수 있으니).
4. `trans = substitute(...)`
   - 전이식을 **미리** 한 프레임 앞당겨 준비
   - 매 프레임 루프에서 바로 `s.add(trans)` 할 수 있게, (x, x', inp)를 (다음 프레임의 x, 다다음 프레임의 x', 다음 프레임의 입력)으로 치환.
   - 치환 매핑:
     - `xns`(이번 프레임의 x') → `ys`(다음 프레임의 x')
     - `xs` (이번 프레임의 x) → `xns`(다음 프레임의 x)
     - `fvs`(이번 프레임 입력) → `nfvs`(다음 프레임 입력)
   - 이렇게 해 두면 **다음 반복에서** `s.add(trans)`를 호출하면 그건 "새 프레임의 전이"가 됨.
5. `goal = substitute(goal, zipp(xs, xns))`
   - `goal(x)`를 `goal(x')`로 **한 프레임 밀기**.
   - 다음 반복에서 `Implies(p, goal)`로 "다음 프레임의 상태에서 goal 시험"이 가능해짐.

6. `xs, xns, fvs = xns, ys, nfvs`
    - 포인터 교체 :
        - 이제 "다음 프레임의 상태"가 **현재 프레임(xs)** 가 되고,
        - 방금 만든 `ys`가 **새로운 다음 상태(xns)** 가 됨.
        - 입력도 마찬가지로 한 칸 전진.

이 과정을 반복하면:

- 반복 0: 초기조건만 있음 → 프레임 0에서 goal 시험
- 반복 1: T(x0,x1) 추가 → 프레임 1에서 goal 시험
- 반복 2: T(x1,x2) 추가 → 프레임 2에서 goal 시험
- ...

즉, 길이 0, 1, 2, ... 로 **경로를 점점 길게 하며** goal 도달 가능성을 확인.

Example 2

문제:
4비트 수에서 $3 + 3 + \cdots + 3 = 10$이 되는 k가 존재하는지 확인하라.

전이 시스템:

- 상태 변수 `x0`이 있으며, 다음 상태는 `x1`
- 초기 상태: `x0 == 0`
- 전이: 매 단계마다 3씩 증가 (`x1 == x0 + 3`)
- 목표: `x0 == 10`

```
x0, x1 = Consts('x0 x1', BitVecSort(4))
bmc(x0 == 0, x1 == x0 + 3, x0 == 10, [], [x0], [x1])
```

결과 해석

- **도달 가능성(reachability)** 을 확인할 수 있지만, **도달 불가능성(non-reachability, safety)**은 증명하지 못함.
- 이러한 **안전성(safety)** 검증은 **IC3 알고리즘**이 담당함.

5.6. Propositional Interpolation (명제적 보간법)

interpolant(보간식) : 두 개의 식 A와 B가 동시에 참일 수 없을 때, A와 B 사이에서 딱 걸쳐있는 중간 식

$A \wedge B$ 가 unsat (모순)일 때,
interpolant I 는 다음 조건을 만족하는 식이다:

1. "$A \models I$ (즉, A가 참이면 I도 반드시 참)"
2. "$I \wedge B \models \bot$ (즉, I와 B를 동시에 만족할 수 없음 → 모순)"
3. "I 에는 A와 B가 공유하는 변수들만 사용됨."

모델(models)과 코어(cores)를 이용해 **interpolant(보간식)** 을 계산할 수 있다 [13].

공식 A, B에 대해 $A \wedge B$가 **unsatisfiable(모순)** 일 때,
보간식 I를 계산하는 절차는 다음과 같이 진행된다:

1. $I = true$ 로 초기화한다.
2. 상태 $[A, B, I]$를 규칙에 따라 점차 포화(saturate)시킨다.

규칙은 다음과 같다:

$$[A, B, I] \Rightarrow [A, B, I \wedge \neg L] \quad \text{if } B \vdash \neg L,\ A \wedge I \nvdash \neg L$$

$$I \quad \text{if } A \vdash \neg I$$

이 규칙에 따라 만들어진 부분 보간식 I는 $B \vdash I$를 만족한다.
알고리즘은 $A \vdash \neg I$가 될 때 종료한다.
조건 $A \wedge I \nvdash \neg L$은 알고리즘이 계속 진행되도록 하며,
각 반복마다 $A \wedge I$의 모델로부터 얻은 **implciant** $L' \supseteq L$ 을 이용해 $A \wedge I$를 확장한다.

예시 코드

```
def mk_lit(m, x):
   if is_true(m.eval(x)):
      return x
   else:
      return Not(x)

def pogo(A, B, xs):
   while sat == A.check():
      m = A.model()
      L = [mk_lit(m, x) for x in xs]
      if unsat == B.check(L):
         notL = Not(And(B.unsat_core()))
         yield notL
         A.add(notL)
      else:
         print("expecting unsat")
         break
```

`pogo(A, B, xs)` : "A는 참이지만 B와는 모순이 나는 구간"을 찾아서, 그 중간 보간식(interpolant) `I`를 점점 정제(refine)해 나감.
즉,

- A : 어떤 한쪽 조건식 (예: 프로그램의 실행조건들)
- B : 반대쪽 조건식 (예: 에러 조건)
- xs : 공유 변수들 (A와 B가 둘 다 쓰는 변수들)

코드 분석

- `mk_lit(m, x)`

```
def mk_lit(m, x):
    if is_true(m.eval(x)):
        return x
    else:
        return Not(x)
```

  - `m`은 A의 모델(즉, A가 SAT일 때의 변수값 할당).
  - `x`가 **True**라면 그냥 `x`, **False**라면 `¬x`를 리터럴로 만듦.
  - 즉, 현재 모델 `m`이 참으로 만드는 **리터럴 집합(L)**을 만들어 주는 함수임.

- `pogo(A, B, xs)` 전체 구조

```
while sat == A.check():
    m = A.model()
    L = [mk_lit(m, x) for x in xs]
    if unsat == B.check(L):
        notL = Not(And(B.unsat_core()))
        yield notL
        A.add(notL)
    else:
        print("expecting unsat")
        break
```

  - `A.check()`
    - A가 SAT인지 확인함.
    - 즉, A가 "아직 가능한 모델"을 가질 동안 반복.
  - `m = A.model()`
    - A의 현재 모델 하나를 가져옴. (즉, A가 만족하는 변수값 조합 하나)
  - `L = [mk_lit(m, x) for x in xs]`
    - A의 모델에서 나온 "현재 참인 리터럴들의 집합" L을 만든다.
    - 예시 : `xs = [x, y, z]`, 모델 `m = {x=True, y=False, z=True}` 라면,

```
L = [x, Not(y), z]
```

- `if unsat == B.check(L):`
  - 이제 **B가 이 L과 함께 모순나는지** 검사함.
  - 즉, A의 현재 모델 `m`이 만들어낸 L을 B에 넣고 SAT를 확인 :
    - B와 L이 **unsat** → B와 A가 충돌하는 영역을 찾음!
    - B와 L이 **sat** → A의 이 모델은 B랑 모순이 아님 → "expecting unsat"
- `B.unsat_core()`
  - B가 **unsat**이면, `B.unsat_core()`는 "B의 어떤 식들이 모순의 원인인지"를 알려줌.
  - 즉, **unsat core = 모순의 최소 부분집합**.
- `notL = Not(And(B.unsat_core()))`
  - unsat core 안의 식들을 전부 AND로 묶고 부정.
  - 즉, "이 모순이 다시는 일어나지 않게 막는" 식을 만듦.
  - 이것이 **interpolant 후보**!
  - 예시 :
    만약 unsat core = `[x, Not(y)]`
    → `And(x, Not(y))`는 모순 원인
    → `Not(And(x, Not(y)))` = `Or(Not(x), y)`
    → 바로 "A와 B 사이를 가르는 중간 식(보간식)"이 된다!
- `yield notL`
  - 보간식 후보를 하나 출력(반환)함.
  - 루프가 계속 돌면 여러 단계의 보간식이 생성될 수 있음.
- `A.add(notL)`
  - "이 보간식 조건을 A에 추가"해서 A를 점점 좁혀감.
  - 즉, 이미 처리한 모순 구간은 제외하고 새 영역을 탐색하게 함.

요약

1. A가 만족 가능한 모델을 하나 찾는다.
2. 그 모델에서 참인 리터럴들의 집합 L을 만든다.
3. B와 L을 같이 놓고 "모순 나는가?"를 확인한다.
4. 만약 모순이면, 그 모순의 원인(`unsat_core`)을 찾아서 그것의 부정(`Not(And(core))`)을 보간식으로 만든다.
5. 그 보간식을 A에 추가해서 A를 점점 정제(refine).
6. A가 SAT일 동안 계속 반복 → 여러 보간식이 생성됨.

Example 3

(reversed) 보간식 예시:
$A : x_1 \neq a_1, a_2 \neq x_2$
$B : x_1 = b_1, b_2 = x_2$

단어집(vocabulary)은 $x_1, x_2, a_1, a_2, b_1, b_2$.
보간식은 B에 의해 함의되고 A와는 모순이다.

```
A = SolverFor("QF_FD")
B = SolverFor("QF_FD")
a1, a2, b1, b2, x1, x2 = Bools('a1 a2 b1 b2 ×1 ×2')
A.add(a1 == x1, a2 != a1, a2 != x2)
B.add(b1 == x1, b2 == b1, b2 == x2)
print(list(pogo(A, B, [x1, x2])))
```

5.7. Monadic Decomposition (단항 분해)

공식 $\varphi[x, y]$가 주어졌다고 하자.
이것을 변수 x, y를 사용하는 **단항(monadic)** 공식들의 조합으로 표현하고자 한다:

$$\varphi(x, y) = \psi_1(x) \wedge \ldots \wedge \psi_k(x) \wedge \theta_1(y) \wedge \ldots \wedge \theta_n(y)$$

여기서 ψ_i와 θ_j는 각각 **하나의 변수만 의존**하는 단항식이다.
이러한 분해를 통해 **extended symbolic transducers** 를
regular symbolic finite transducers 로 변환할 수 있다.
이는 분석과 최적화에 유용하다.
Z3에서는 [60]에서 개발된 **monadic decomposition** 절차를 다음과 같이 구현할 수 있다.

구현 코드

```
from z3 import *

def nu_ab(R, x, y, a, b):
    x_ = [Const("x_%d" % i, x[i].sort()) for i in range(len(x))]
    y_ = [Const("y_%d" % i, y[i].sort()) for i in range(len(y))]
    return Or(Exists(y_, R(x+y_) != R(a+y_)), Exists(x_, R(x_+y) != R(x_+b)))

def isUnsat(fml):
    s = Solver()
    s.add(fml)
    return s.check() == unsat

def lastSat(s, m, fmls):
    if len(fmls) == 0: return m
    s.push()
    s.add(fmls[0])
```

```
    if s.check() == sat:
        m = lastSat(s, s.model(), fmls[1:])
    s.pop()
    return m

def mondec(R, variables):
    print(variables)
    phi = R(variables)
    if len(variables) == 1: return phi
    l = int(len(variables)/2)
    x, y = variables[:l], variables[l:]
    def dec(nu, pi):
        if isUnsat(And(pi, phi)): return BoolVal(False)
        if isUnsat(And(pi, Not(phi))): return BoolVal(True)
        fmls = [BoolVal(True), phi, pi]
        # try to extend nu
        m = lastSat(nu, None, fmls)
        assert(m != None)
        a = [m.evaluate(z, True) for z in x]
        b = [m.evaluate(z, True) for z in y]
        psi_a = And(R(a+y), R(x+b))
        phi_a = mondec(lambda z: R(a+z), y)
        phi_b = mondec(lambda z: R(z+b), x)
        nu.push()
        nu.add(nu_ab(R, x, y, a, b))
        t = dec(nu, And(pi, psi_a))
        f = dec(nu, And(pi, Not(psi_a)))
        nu.pop()
        return If(And(phi_a, phi_b), t, f)
    return dec(Solver(), BoolVal(True))
```

💡 **핵심 개념**

- `R(variables)` : 분해하고 싶은 원래의 조건식(불리언). 예: `R([x,y]) = y>0 ∧ power_of_two(y) ∧ ((x & (y % M)) ≠ 0)` 같은 형태.
- **Monadic(단항)**: 식을 `ψ_i(x들 중 일부 하나만)` 또는 `θ_j(y들 중 하나만)` 처럼 **한 변수만** 의존하는 식들의 AND/OR 조합으로 표현하려는 목표.

함수별 역할

- `nu_ab(R, x, y, a, b)`

```
def nu_ab(R, x, y, a, b):
    x_ = [Const("x_%d" % i, x[i].sort()) for i in range(len(x))]
    y_ = [Const("y_%d" % i, y[i].sort()) for i in range(len(y))]
    return Or(
        Exists(y_, R(x+y_) != R(a+y_)),
        Exists(x_, R(x_+y) != R(x_+b))
    )
```

  - 변수 집합을 `x`와 `y` 두 덩어리로 나눴을 때, 샘플 포인트 `a` (x값들), `b` (y값들)에 대해
    - "x를 `a`로 고정했을 때 y를 바꾸면 결과가 달라질 수 있는가?"
    - "y를 `b`로 고정했을 때 x를 바꾸면 결과가 달라질 수 있는가?"
  - 위 둘 중 **하나라도 가능**하면 `Or(...)`가 **참**이 됨.
  - 즉, `nu_ab`는 "현재 고정점 `(a, b)` 주변에서 R의 값이 **x-축 또는 y-축으로 변화를 보일 여지**가 있는가?"를 나타내는 **구분(분기)용 제약**을 만든다.

- `isUnsat(fml)`

```
def isUnsat(fml):
    s = Solver(); s.add(fml)
    return s.check() == unsat
```

  - 말 그대로 "`fml`이 UNSAT인가?"를 빠르게 판단.

- `lastSat(s, m, fmls)`

```
def lastSat(s, m, fmls):
    if len(fmls) == 0: return m
    s.push(); s.add(fmls[0])
    if s.check() == sat:
        m = lastSat(s, s.model(), fmls[1:])
    s.pop()
    return m
```

  - `fmls` 리스트를 **앞에서부터 가능한 만큼** 누적하며 SAT을 유지하는 "마지막 모델"을 얻는다.
  - 쉽게 말해, `True → phi → pi` 순서로 붙여보며 **가능한 끝까지 간 모델** `m` 을 가져온다.
    - 여기서 `phi = R(variables)` (원래 식), `pi`는 누적 분기조건(아래 설명).

- `mondec(R, variables)`

```
def mondec(R, variables):
    phi = R(variables)
    if len(variables) == 1: return phi          # 단항이면 끝
    l = int(len(variables)/2)
    x, y = variables[:l], variables[l:]         # 반으로 쪼갬
    def dec(nu, pi):
        if isUnsat(And(pi, phi)):     return BoolVal(False)
        if isUnsat(And(pi, Not(phi))):return BoolVal(True)
        fmls = [BoolVal(True), phi, pi]
        m = lastSat(nu, None, fmls)             # 가능한 멀리까지의 모델
        assert(m != None)
        a = [m.evaluate(z, True) for z in x]    # 현재 모델에서의 x-값 스냅샷
        b = [m.evaluate(z, True) for z in y]    # 현재 모델에서의 y-값 스냅샷
        psi_a = And(R(a+y), R(x+b))             # (a, y) & (x, b) 둘 다에서 참
        phi_a = mondec(lambda z: R(a+z), y)     # x를 a로 고정해 y-부분 분해
        phi_b = mondec(lambda z: R(z+b), x)     # y를 b로 고정해 x-부분 분해
        nu.push()
        nu.add(nu_ab(R, x, y, a, b))            # (a,b) 주변에서 바뀔 여지 강제
        t = dec(nu, And(pi, psi_a))             # (둘 다에서 참)인 영역 분기
        f = dec(nu, And(pi, Not(psi_a)))        # 그 밖의 영역 분기
        nu.pop()
        return If(And(phi_a, phi_b), t, f)      # 양쪽이 단항적으로 잘 분해되면 t, 아니면 f 쪽
    return dec(Solver(), BoolVal(True))
```

  - **Base case**: 변수 한 개만 남으면 이미 "단항" → `phi` 그대로 반환.
  - **Divide**: 변수들을 반으로 나눠 `x` / `y` 로 재귀적으로 처리.
  - `dec(nu, pi)` : 실제 분해를 수행하는 내부 재귀.
    - `nu` : 보조 솔버(분기/증분 제약 쌓는 용도)
    - `pi` : 지금까지 축적된 "분기 조건"(영역을 점점 쪼개는 조건)
    - 빠른 단정:
      - `pi ∧ phi` 가 UNSAT → 이 영역에서는 R가 **항상 거짓** → `False`
      - `pi ∧ ¬phi` 가 UNSAT → 이 영역에서는 R가 **항상 참** → `True`
    - 둘 다 아니면(아직 영역이 혼재) → 현재 가능한 모델 `m` 을 `lastSat` 로 얻고, 그 모델의
      - `a` : 현재 x 값들
      - `b` : 현재 y 값들을 뽑아 고정점으로 사용.
    - `psi_a = R(a,y) ∧ R(x,b)` :
      - "x를 a로 고정했을 때도 참이고, y를 b로 고정했을 때도 참"인 지점.
      - 이걸 경계로 **영역을 두 갈래**(t-브랜치/f-브랜치)로 분할.
    - `phi_a = mondec(lambda z: R(a+z), y)` :
      - x를 a로 고정한 상태에서 y에 대해서만 단항 분해 재귀.

- `phi_b = mondec(lambda z: R(z+b), x)` :
  - y를 b로 고정한 상태에서 x에 대해서만 단항 분해 재귀.
- `nu.add(nu_ab(..., a, b))` :
  - 방금 잡은 고정점 `(a,b)` 주변에서 **값이 바뀔 수 있는 자유도**(x-방향 혹은 y-방향)를 강제하여, 다음 호출 때 **의미 있는 분할**이 되도록 유도.
- `t = dec(..., pi ∧ psi_a)` , `f = dec(..., pi ∧ ¬psi_a)` :
  - 두 영역을 재귀로 더 쪼갠 결과.
- `If(And(phi_a, phi_b), t, f)` :
  - 만약 "x를 a로 고정했을 때의 분해(`phi_a`)"와 "y를 b로 고정했을 때의 분해(`phi_b`)"가 **둘 다 단항 분해로 충분**하다면, 더 제약적인 쪽(t)을 채택하고,
    그렇지 않으면 f-분기(보완 영역)를 택한다.
  - 즉, **양쪽 축에서 단항 분해가 깔끔하면 그 경계(psi_a) 안쪽 표현(t)을 쓰고, 아니면 바깥(f) 쪽을 쓴다**는 선택 로직.

Example 4

비트 벡터 x, y (각각 2k비트) 에 대해 monadic decomposition을 수행하는 예제:

$$y > 0 \wedge (y \& (y-1)) = 0 \wedge (x \& (y \% ((1 << k) - 1))) \neq 0$$

이를 코드로 나타내면:

```
def test_mondec(k):
   R = lambda v: And(v[1] > 0, (v[1] & (v[1]-1)) == 0,
                ((v[0] & (v[1] % ((1 << k) - 1))) != 0))
   bvs = BitVecSort(2*k)
   x, y = Consts('x y', bvs)
   res = mondec(R, [x, y])
   assert(isUnsat(res != R([x, y])))
   print("mondec(", R([x, y]), ") =", res)

test_mondec(2)
```

5.8. Subterm Simplification (부분항 단순화)

Z3의 기본 **simplifier(단순화기)** 는 단순한 **대수적 단순화**만 수행하며,
문맥(context) 정보를 활용하지 않는다.

예를 들어,
다음 식에서

$$t = 4 + 4 * (((H - 1)/2)/2)$$

는 H와 동일하다는 정보를 solver가 알고 있다고 하자.
이 정보를 이용해 더 똑똑한 simplifier를 만들 수 있다.

예시 코드

```
H = Int('H')
s = Solver()
t = 4 + 4 * (((H - 1) / 2) / 2)
s.add(H % 4 == 0)
s.check()
m = s.model()
print(t, "→", simplify(s, m, t))
```

부분항(subterm) 추출 함수

하나의 식에서 중복 탐색을 피하면서 부분항을 찾는 함수:

```
def subterms(t):
    seen = {}
    def subterms_rec(t):
        if is_app(t):
            for ch in t.children():
                if ch in seen:
                    continue
                seen[ch] = True
                yield ch
                yield from subterms_rec(ch)
    return { s for s in subterms_rec(t) }
```

단순화 루틴 정의

1. **are_equal(s, t1, t2)**: 두 항이 같은지 solver로 검사
2. **simplify(slv, mdl, t)**:
   모델과 문맥정보를 이용해 `t`의 하위항(subterm)을 찾아 동일한 항으로 대체

```
def are_equal(s, t1, t2):
    s.push()
    s.add(t1 != t2)
    r = s.check()
    s.pop()
```

```
    return r == unsat

def simplify(slv, mdl, t):
    subs = subterms(t)
    values = { s: mdl.eval(s) for s in subs }
    values[t] = mdl.eval(t)
    def simplify_rec(t):
        subs = subterms(t)
        for s in subs:
            if s.sort().eq(t.sort()) and values[s].eq(values[t]) and are_equal(slv, s, t):
                return simplify_rec(s)
        chs = [simplify_rec(ch) for ch in t.children()]
        return t.decl()(chs)
    return simplify_rec(t)
```