

Problem Solving using SMT Solver (2)

COSE419, Spring 2024

Hakjoo Oh

Due: 5/10 23:59

Problem 1 (Program Verification) 프로그램 검증(*Program Verification*)은 프로그램의 구현(*Implementation*)이 명세(*Specification*)를 만족하는지 확인하는 기술이다. 이번 문제에서는 라이브러리 함수로 구성된 프로그램이 입출력 예제로 주어진 명세를 만족하는지 확인하는 검증기를 구현해보자. 프로그램은 다음과 같이 하나의 함수로 주어진다 하자. 함수의 몸통은 반복문/제어문을 포함하지 않는 연속된 라이브러리 함수 호출들로 구성된다.

```
def f(x):
    y := add(x, x); // y = x + x
    z := mul(y, y); // z = y × y
    return z
```

(1)

입력이 x 일때 출력으로 $4x^2$ 을 계산하는 프로그램을 라이브러리 함수 *add*와 *mul*을 각각 한번씩 호출하여 구현한 프로그램이다. 라이브러리 함수의 의미는 입출력 변수들의 관계를 나타내는 일차 논리식으로 주어진다. 위의 예제에서 주석으로 적혀있는 $y = x + x$, $z = y \times y$ 가 각각 라이브러리 함수 *add*와 *mul*의 의미를 뜻한다. 프로그램 명세는 다음과 같이 하나의 입출력 예제로 주어진다.

$$3 \mapsto 36$$

프로그램의 입력이 3일 때 ($x = 3$) 반환값은 36이어야 함을 뜻한다.

Problem Definition 검증기는 두 가지 입력을 받는다.

1. 프로그램 구현:

```
def f(I):
    Y1 := f1( $\vec{X}_1$ ); //  $\phi_1(\vec{X}_1, Y_1)$ 
    ⋮
    YN := fN( $\vec{X}_N$ ); //  $\phi_N(\vec{X}_N, Y_N)$ 
    return YN
```

- I : 프로그램의 입력 변수¹
- f_i : 라이브러리 함수 이름
- Y_i : 라이브러리 f_i 의 출력 변수
- \vec{X}_i : 라이브러리 f_i 의 인자들. 인자는 1개 이상이고 임의의 식이 아닌 변수만 올 수 있다.
- ϕ_i : 라이브러리 f_i 의 의미를 명세하는 일차 논리식. 입출력 변수들(\vec{X}_i, Y_i)의 관계를 뜻한다. ϕ 는 아래 문법으로 정의되는 간단한 논리식으로 기술된다고 하자:

$$\phi \rightarrow e = e, \quad e \rightarrow n \in \mathbb{Z} \mid x \mid e + e \mid e \times e$$

예를 들어, $y = x + x$, $z = y \times y$, $y + 1 = x$, $z \times 2 = x + y$ 등을 표현할 수 있는 논리식이다.

¹프로그램의 입력은 1개 이상일 수 있다. 문제 정의에서는 편의상 1개라고 가정한다.

2. 프로그램 명세:

$$v_1 \mapsto v_2$$

하나의 입출력 예제로 주어진다. v_1, v_2 는 모두 정수이다.

검증기의 목표는 주어진 프로그램이 입출력 명세를 만족하는지 증명하는 것이다:

$$I = v_1 \wedge \bigwedge_{i=1}^N \phi_i(\vec{X}_i, Y_i) \implies Y_N = v_2$$

Implementation 프로그램, 라이브러리, 입출력 명세의 타입을 다음과 같이 정의하였다.

```
type pgm = var list * lib list * var
and lib = var * var list * var * phi
and var = string
and phi = EQ of exp * exp
and exp =
  | INT of int
  | VAR of var
  | ADD of exp * exp
  | MUL of exp * exp
type spec = int list * int
```

예를 들어, 프로그램 (1)은 다음과 같이 표현된다:

```
(["x"],
 [("add", ["x"; "x"], "y", EQ (VAR "y", ADD (VAR "x", VAR "x")));
  ("mul", ["y"; "y"], "z", EQ (VAR "z", MUL (VAR "y", VAR "y")))],
 "z")
```

입출력 명세는 다음과 같이 표현된다:

```
([2], 16)
```

주어진 프로그램이 입출력 명세를 만족하는지 확인하는 함수 `verify`를 작성하시오.

```
verify : pgm -> spec -> bool
```

검증에 성공할 경우 `true`를 반환한다.

Examples

- 위 예제는 다음과 같이 실행할 수 있다.

```
verify
  (["x"],
   [("add", ["x"; "x"], "y", EQ (VAR "y", ADD (VAR "x", VAR "x")));
    ("mul", ["y"; "y"], "z", EQ (VAR "z", MUL (VAR "y", VAR "y")))],
   "z")
  ([2], 16)
```

- 프로그램은 여러 입력을 가질 수 있다. 아래 프로그램

```
def f(x0,x1,x2,x3):
  o1 := mul(x0,x2) // o1 = x0 * x2
  o2 := mul(x2,o1) // o2 = x2 * o1
  o3 := add(o2,o2) // o3 = o2 + o2
  o4 := add(o2,o3) // o4 = o2 + o3
  return o4
```

이 입출력 명세 $(1, 2, 3, 4) \mapsto 27$ 을 만족하는지 여부를 다음과 같이 확인할 수 있다.

```
verify (
  ["x0"; "x1"; "x2"; "x3"],
  [
    ("mul", ["x0"; "x2"], "o1", EQ (VAR "o1", MUL (VAR "x0", VAR "x2")));
    ("mul", ["x2"; "o1"], "o2", EQ (VAR "o2", MUL (VAR "x2", VAR "o1")));
    ("add", ["o2"; "o2"], "o3", EQ (VAR "o3", ADD (VAR "o2", VAR "o2")));
    ("add", ["o2"; "o3"], "o4", EQ (VAR "o4", ADD (VAR "o2", VAR "o3")));
  ],
  "o4"
)
([1; 2; 3; 4], 27)
```

- 함수 $f(x) = (x - 1)/2$ 는 아래와 같이 구현할 수 있다.

```
def f(x):
  o1 := sub1(x) // o1 + 1 = x
  o2 := div2(o1) // o2 * 2 = o1
  return o2
```

입출력 명세 $5 \mapsto 2$ 를 만족하는지 아래와 같이 확인할 수 있다.

```
verify
  (["x"],
  [
    ("sub1", ["x"], "o1", EQ (ADD (VAR "o1", INT 1), VAR "x"));
    ("div2", ["o1"], "o2", EQ (MUL (VAR "o2", INT 2), VAR "o1"));
  ],
  "o2"
)
([5], 2)
```

Problem 2 (Program Synthesis) 프로그램 합성(*Program Synthesis*)은 명세로부터 코드를 자동으로 생성하는 기술이다. 프로그램 검증의 역방향이다. 사용 가능한 라이브러리 함수들과 입출력 명세를 받아서 프로그램을 자동 합성하는 함수 `synthesize`를 작성하시오.

```
synthesize : lib list -> spec -> pgm option
```

Examples

- 합성기 `synthesize`를 사용하여 프로그램 (1)을 자동 생성해 보자:

```
synthesize [
  ("mul", ["i11"; "i12"], "o1", EQ (VAR "o1", MUL (VAR "i11", VAR "i12")));
  ("add", ["i21"; "i22"], "o2", EQ (VAR "o2", ADD (VAR "i21", VAR "i22")));
] ([3], 36)
```

`synthesize`의 첫번째 인자로 사용 가능한 라이브러리 함수 정의들이 주어진다. 라이브러리 `mul`과 `add`를 사용할 수 있음을 뜻하고 각 라이브러리 함수의 의미가 함께 주어진다. 예를 들어, `mul`은 입출력 변수들을 각각 `i11`, `i12` 및 `o1`이라고 할 때 $o1 = i11 \times i12$ 를 만족하는 라이브러리임을 뜻한다. 주어진 라이브러리 함수들에서 사용된 입출력 변수들(`i11`, `i12`, `o1`, `i21`, `i22`, `o2`)은 모두 유일한 이름을 가진다고 하자. `synthesize`의 두번째 인자는 명세로 주어진 입력값([3]) 및 출력값(36)이다. 실행결과는 다음과 같다:

```
Some (["x"],
  [("add", ["x"; "x"], "y", EQ (VAR "y", ADD (VAR "x", VAR "x")));
   ("mul", ["y"; "y"], "z", EQ (VAR "z", MUL (VAR "y", VAR "y")))],
  "z")
```

텍스트로 표현하면 다음과 같다:

```
def f(x):
  y := add(x,x)
  z := mul(y,y)
  return z
```

의미가 같다면 프로그램이 사용하는 변수들의 이름은 달라도 된다. 예를 들어, 아래와 같이 라이브러리 함수 정의에서 사용된 변수들을 재사용해도 된다.

```
def f(x):
  o2 := add(x,x)
  o1 := mul(o2,o2)
  return o1
```

```
Some (["x"],
  [("add", ["x"; "x"], "o2", EQ (VAR "o2", ADD (VAR "x", VAR "x")));
   ("mul", ["o2"; "o2"], "o1", EQ (VAR "o1", MUL (VAR "o2", VAR "o2")))],
  "o1")
```

생성된 프로그램이 올바른지 확인하기 위해 Problem 1에서 구현한 `verify`를 이용할 수 있다:

```
verify (["x"],
  [("add", ["x"; "x"], "y", EQ (VAR "y", ADD (VAR "x", VAR "x")));
   ("mul", ["y"; "y"], "z", EQ (VAR "z", MUL (VAR "y", VAR "y")))],
  "z") ([3], 36)
```

```
verify (["x"],
  [("add", ["x"; "x"], "o2", EQ (VAR "o2", ADD (VAR "x", VAR "x")));
   ("mul", ["o2"; "o2"], "o1", EQ (VAR "o1", MUL (VAR "o2", VAR "o2")))],
  "o1") ([3], 36)
```

- 문제를 간단히 하기 위해서 입력으로 주어진 라이브러리 함수들은 프로그램에서 최소/최대 한 번씩 사용된다고 가정하자. 예를 들어, 곱셈을 세 번 사용하려면 세 개의 곱셈 라이브러리를 제공해 주어야 한다:

```
synthesize [
  ("mul", ["i11"; "i12"], "o1", EQ (VAR "o1", MUL (VAR "i11", VAR "i12")));
  ("mul", ["i21"; "i22"], "o2", EQ (VAR "o2", MUL (VAR "i21", VAR "i22")));
  ("mul", ["i31"; "i32"], "o3", EQ (VAR "o3", MUL (VAR "i31", VAR "i32")));
] ([2], 256)
```

결과는 $f(x) = x^8$ 을 뜻하는 아래 프로그램이다.

```
def f(x):
  o1 := mul(x,x)
  o2 := mul(o1,o1)
  o3 := mul(o2,o2)
  return o3
```

주어진 라이브러리는 반드시 사용된다고 가정한다. 이 조건을 만족하기 위해, 필요하지 않은 라이브러리가 주어진 경우, 사용되지 않는 코드(*dead code*)로 포함시키면 된다. 예를 들어, 위 예제의 명세를 아래와 같이 주었다고 해 보자.

```

synthesize [
  ("mul", ["i11"; "i12"], "o1", EQ (VAR "o1", MUL (VAR "i11", VAR "i12")));
  ("mul", ["i21"; "i22"], "o2", EQ (VAR "o2", MUL (VAR "i21", VAR "i22")));
  ("mul", ["i31"; "i32"], "o3", EQ (VAR "o3", MUL (VAR "i31", VAR "i32")));
  ("add", ["i41"; "i42"], "o4", EQ (VAR "o4", ADD (VAR "i41", VAR "i42")));
] ([2], 256)

```

이 경우 다음과 같이 불필요한 라이브러리(add)는 결과값에 영향을 미치지 않도록 *dead code*의 형태로 포함시킬 수 있다.

```

def f(x):
  y := mul(x,x)
  z := mul(y,y)
  p := mul(z,z)
  q := add(x,x) // dead code (q is not used in the rest of the program)
  return p

```

또는 아래와 같이 *dead code*없이 주어진 모든 라이브러리를 사용하면서 입출력 명세를 여전히 만족하는 프로그램을 생성해도 된다.

```

def f(x):
  y := mul(x,x) // x^2
  z := add(y,y) // 2x^2
  p := mul(y,z) // 2x^4
  q := mul(p,z) // 4x^6
  return q

```

이 경우 $f(x) = 4x^6$ 을 계산하는 프로그램을 합성했고, 이 또한 주어진 입출력 예제($3 \mapsto 256$)를 만족시킨다.²

- 입력 변수는 여러개일 수 있다.

```

synthesize [
  ("add", ["i11"; "i12"], "o1", EQ (VAR "o1", ADD (VAR "i11", VAR "i12")));
]
([2; 3], 5)

def f(x0,x1):
  o1 := add(x0,x1)
  return o1

```

- 주어진 라이브러리 함수들로 입출력 명세를 만족하는 프로그램을 생성할 방법이 없는 경우 합성에 실패하고 *None*을 반환한다. 예를 들어, 아래 실행 결과는 *None*이어야 한다. 함수 $f(x) = x^8$ 는 곱셈 두 개만 써서는 구현할 수 없기 때문이다.

```

synthesize [
  ("mul", ["i11"; "i12"], "o1", EQ (VAR "o1", MUL (VAR "i11", VAR "i12")));
  ("mul", ["i21"; "i22"], "o2", EQ (VAR "o2", MUL (VAR "i21", VAR "i22")));
] ([2], 256)

```

²이와 같이 입출력 예제로 프로그램 명세를 기술하는 경우 일반적으로 생성하고자 하는 프로그램의 의미($f(x) = x^8$)를 온전히 표현할 수 없다. 따라서 주어진 입출력 명세는 만족하지만 의도와는 다른 프로그램이 생성될 수 있다. 귀납적 학습(Inductive learning)에서 흔히 발생하는 과적합(Overfitting) 문제이다. 과적합된 프로그램도 정답으로 처리된다.

Problem Definition 합성기는 두 가지 입력을 받는다.

1. 프로그램 명세:

$$v_1 \mapsto v_2$$

문제 정의에서는 프로그램의 입력 변수가 한 개인 경우만 생각하자(구현에서는 1개 이상의 일반적 경우들을 고려해야 한다). 입출력 값은 정수이다 ($v_1, v_2 \in \mathbb{Z}$).

2. 라이브러리 함수들:

$$\{\langle f_i, \vec{I}_i, O_i, \phi_i(\vec{I}_i, O_i) \rangle \mid i = 1, \dots, N\}$$

- N : 주어진 라이브러리 함수 개수 ($N \geq 1$)
- f_i : 라이브러리 함수 이름
- \vec{I}_i : 라이브러리 함수의 인자들 (1개 이상)
- O_i : 라이브러리 함수의 출력 변수
- $\phi_i(\vec{I}_i, O_i)$: 라이브러리의 명세를 기술하는 일차 논리식

위와 같은 입력이 주어졌을때, 합성기의 목표는 주어진 N 개의 라이브러리를 한번씩 모두 사용하면서 입출력 예제를 만족하는 프로그램을 생성하는 것이다.

SMT Encoding 프로그램 합성에는 크게 두 가지 방식이 있다: (1) 주어진 명세를 만족하는 프로그램이 찾아질 때까지 모든 가능한 프로그램들을 나열 탐색(Enumerative search)하는 방식. (2) 프로그램들의 전체 공간을 논리식으로 표현한 후 명세를 만족하는 프로그램이 존재하는지 여부(Satisfiability)를 판단하는 방식. 두 번째 방식으로 가 보자.

합성할 프로그램은 일반적으로 다음과 같이 쓸 수 있다:

$$\begin{aligned} \text{def } f(I) : \\ & O_{\pi_1} := f_{\pi_1}(\vec{V}_{\pi_1}); \\ & \vdots \\ & O_{\pi_N} := f_{\pi_N}(\vec{V}_{\pi_N}); \\ \text{return } O_{\pi_N} \end{aligned} \tag{2}$$

- I : 프로그램의 입력 변수
- π_1, \dots, π_N : $1, \dots, N$ 의 순열(permutation). 각 라이브러리를 한번씩 사용한다는 뜻이다.
- \vec{V}_{π_i} : 라이브러리 함수를 호출하는데 사용되는 실제 인자(actual argument)들. \vec{V}_{π_i} 내의 각 변수는 입력 변수(I)이거나 부품들의 출력 변수들 O_{π_j} ($j < i$) 가운데 하나이다.

위와 같은 프로그램들의 전체 집합을 일차 논리식으로 표현해보자. 이를 위해 정수 타입의 위치 변수(location variable) $l_x \in \mathbb{Z}$ 를 프로그램과 주어진 라이브러리의 입력 및 출력 변수들마다 하나씩 도입한다 [1]. 위치 변수들의 전체 집합 L 은 다음과 같이 정의된다:

$$L = \{l_x \mid x \in \mathcal{I} \cup \mathcal{O} \cup \{I, O\}\}$$

여기서 \mathcal{I} 와 \mathcal{O} 는 각각 라이브러리들의 입력 및 출력 변수들의 전체 집합이고,

$$\mathcal{I} = \bigcup_{i=1}^N \vec{I}_i, \quad \mathcal{O} = \{O_1, O_2, \dots, O_N\}$$

\mathcal{O} 는 프로그램의 출력값을 뜻하기 위해 도입한 변수로 $O = O_{\pi_N}$ 가 성립한다.

- x 가 O_i (라이브러리 f_i 의 출력 변수)를 지칭하는 경우, l_x 는 라이브러리 함수 f_i 가 합성된 프로그램에서 자리하는 위치를 뜻한다. 예를 들어, $l_{O_2} = 3$ 이라면 라이브러리 f_2 가 합성될 프로그램에서 세 번째 라인에 위치함을 뜻한다.

- x 가 \vec{l}_{ij} (라이브러리 f_i 의 j 번째 입력 변수)를 지칭하는 경우, l_x 는 라이브러리 함수 f_i 의 j 번째 인자가 정의된 프로그램 위치를 뜻한다. 예를 들어, $l_{\vec{l}_{12}} = 3$ 은 f_1 의 두 번째 인자가 라인 3에서 정의된 변수임을 뜻하고 $l_{\vec{l}_{21}} = 1$ 은 f_2 의 첫 번째 인자가 라인 1에서 정의된 변수임을 뜻한다.
- $l_I = 0$ 으로 정의한다. 프로그램 입력 변수의 위치를 0으로 정의한다는 뜻이다.
- $l_O = N$ 으로 정의한다. $O = O_{\pi_N}$ 이 성립한다는 뜻이다.

올바른 프로그램(well-formed program)을 나타내기 위해 위치 변수는 아래 조건들을 만족해야 한다:

- 프로그램 및 라이브러리의 입력 및 출력 변수들의 위치 범위:

$$l_I = 0 \wedge l_O = N \wedge \left(\bigwedge_{x \in \mathcal{I}} 0 \leq l_x \leq N \right) \wedge \left(\bigwedge_{x \in \mathcal{O}} 1 \leq l_x \leq N \right)$$

- 같은 위치에 두 개의 라이브러리 함수가 위치해서는 안된다. f_i 의 위치를 l_{O_i} 로 나타내기로 했으므로 다음과 같이 표현할 수 있다:

$$\bigwedge_{O_i, O_j \in \mathcal{O}} i \neq j \rightarrow l_{O_i} \neq l_{O_j}$$

- 라이브러리 함수의 인자로 사용되는 변수들은 이미 값이 정의된 것들이어야 한다. x 가 f_i 의 입력 변수인 경우($x \in \vec{l}_i$) l_x 는 해당 인자에 주어지는 변수가 정의된 위치를 나타내기로 했으므로 f_i 의 위치(l_{O_i})보다 이전이어야 한다 ($l_x < l_{O_i}$):

$$\bigwedge_{i=1}^N \bigwedge_{x \in \vec{l}_i} l_x < l_{O_i}$$

정리하면 (2)와 같은 형태를 가지는 프로그램의 전체 공간은 논리식 ψ_{wfp} 로 나타낼 수 있다:

$$\begin{aligned} \psi_{wfp} := & l_I = 0 \wedge l_O = N \wedge \left(\bigwedge_{x \in \mathcal{I}} 0 \leq l_x \leq N \right) \wedge \left(\bigwedge_{x \in \mathcal{O}} 1 \leq l_x \leq N \right) \wedge \\ & \left(\bigwedge_{O_i, O_j \in \mathcal{O}} i \neq j \rightarrow l_{O_i} \neq l_{O_j} \right) \wedge \left(\bigwedge_{i=1}^N \bigwedge_{x \in \vec{l}_i} l_x < l_{O_i} \right) \end{aligned}$$

ψ_{wfp} 는 위치 변수들(L)에 대한 일차 논리식이다. ψ_{wfp} 를 참으로 만드는 해($L \rightarrow \mathbb{Z}$ 타입의 매핑)들의 집합을 아래와 같이 정의할 수 있고,

$$\mathbb{P} = \{M \mid M \in L \rightarrow \mathbb{Z}, M \models \psi_{wfp}\}$$

각각의 해 M 은 (2)와 같은 형태의 프로그램 하나를 결정하므로 \mathbb{P} 는 전체 프로그램 공간을 나타낸다. 이제 공간 \mathbb{P} 에 주어진 입출력 명세를 만족하는 프로그램이 존재하는지 확인해보자:

- 프로그램을 “실행”하기 위해서 프로그램내의 데이터 흐름을 연결해 주어야 한다. 위치 변수의 의미를 생각하면 아래와 같이 변수들의 연결 관계(Connectivity)가 성립해야 함을 알 수 있다:

$$\psi_{conn} := \bigwedge_{x, y \in \mathcal{I} \cup \mathcal{O} \cup \{I, O\}} (l_x = l_y \rightarrow x = y)$$

어떤 두 변수 x, y 의 위치가 같다면($l_x = l_y$), x, y 는 동일한 값을 가져야 한다는 뜻이다. 예를 들어, $l_{O_1} = l_{\vec{l}_{31}}$ 이면 $O_1 = \vec{l}_{31}$ 이어야 한다. f_3 의 첫 번째 인자 값이 f_1 의 출력 변수로부터 전달됨을 의미한다.

- 주어진 라이브러리들의 의미가 모두 성립함을 표현한다:

$$\psi_{lib} := \bigwedge_{i=1}^N \phi_i(\vec{l}_i, O_i)$$

- 입출력 명세가 만족됨을 표현한다:

$$\psi_{io} := I = v_1 \wedge O = v_2$$

아래 논리식이 참이 될 수 있다면(*Satisfiable*) 주어진 명세를 만족하는 프로그램이 존재하게 된다:

$$\psi_{wfp} \wedge \psi_{conn} \wedge \psi_{lib} \wedge \psi_{io}$$

Example

- *Libraries and specification provided:*

$$\{\langle f_1, [I_{11}], O_1, O_1 + 1 = I_{11} \rangle, \langle f_2, [I_{21}, I_{22}], O_2, O_2 = I_{21} \times I_{22} \rangle\}, \quad 5 \mapsto 16$$

- $\mathcal{I}, \mathcal{O}, L$:

$$\mathcal{I} = \{I_{11}, I_{21}, I_{22}\}, \quad \mathcal{O} = \{O_1, O_2\}, \quad L = \{l_I, l_O, l_{I_{11}}, l_{I_{21}}, l_{I_{22}}, l_{O_1}, l_{O_2}\}$$

- ψ_{wfp} :

$$l_I = 0 \wedge l_O = 2 \wedge 0 \leq l_{I_{11}} \leq 2 \wedge 0 \leq l_{I_{21}} \leq 2 \wedge 0 \leq l_{I_{22}} \leq 2 \wedge 1 \leq l_{O_1} \leq 2 \wedge 1 \leq l_{O_2} \leq 2 \wedge \\ l_{O_1} \neq l_{O_2} \wedge \\ l_{I_{11}} < l_{O_1} \wedge l_{I_{21}} < l_{O_2} \wedge l_{I_{22}} < l_{O_2}$$

- ψ_{conn} :

$$\begin{array}{llll} l_I = l_I \rightarrow I = I & \wedge & l_O = l_I \rightarrow O = I & \wedge & l_{O_2} = l_I \rightarrow O_2 = I & \wedge \\ l_I = l_O \rightarrow I = O & \wedge & l_O = l_O \rightarrow O = O & \wedge & l_{O_2} = l_O \rightarrow O_2 = O & \wedge \\ l_I = l_{I_{11}} \rightarrow I = I_{11} & \wedge & l_O = l_{I_{11}} \rightarrow O = I_{11} & \wedge & l_{O_2} = l_{I_{11}} \rightarrow O_2 = I_{11} & \wedge \\ l_I = l_{I_{21}} \rightarrow I = I_{21} & \wedge & l_O = l_{I_{21}} \rightarrow O = I_{21} & \wedge & \dots & l_{O_2} = l_{I_{21}} \rightarrow O_2 = I_{21} & \wedge \\ l_I = l_{I_{22}} \rightarrow I = I_{22} & \wedge & l_O = l_{I_{22}} \rightarrow O = I_{22} & \wedge & l_{O_2} = l_{I_{22}} \rightarrow O_2 = I_{22} & \wedge \\ l_I = l_{O_1} \rightarrow I = O_1 & \wedge & l_O = l_{O_1} \rightarrow O = O_1 & \wedge & l_{O_2} = l_{O_1} \rightarrow O_2 = O_1 & \wedge \\ l_I = l_{O_2} \rightarrow I = O_2 & \wedge & l_O = l_{O_2} \rightarrow O = O_2 & \wedge & l_{O_2} = l_{O_2} \rightarrow O_2 = O_2 & \end{array}$$

- ψ_{lib} :

$$O_1 + 1 = I_{11} \wedge O_2 = I_{21} \times I_{22}$$

- ψ_{io} :

$$I = 5 \wedge O = 16$$

- M such that $M \models \psi_{wfp} \wedge \psi_{conn} \wedge \psi_{lib} \wedge \psi_{io}$:

$$M = \left\{ \begin{array}{ll} I & = 5, \\ O & = 16, \\ I_{11} & = 5, \\ I_{21} & = 4, \\ I_{22} & = 4, \\ O_1 & = 4, \\ O_2 & = 16, \end{array} \quad \begin{array}{ll} l_I & = 0 \\ l_O & = 2 \\ l_{I_{11}} & = 0 \\ l_{I_{21}} & = 1 \\ l_{I_{22}} & = 1 \\ l_{O_1} & = 1 \\ l_{O_2} & = 2 \end{array} \right\}$$

- *Program that M denotes:*

```
def f(I):
    O1 := f1(I);
    O2 := f2(O1, O1);
    return O2
```


References

- [1] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 62–73, New York, NY, USA, 2011. Association for Computing Machinery.