Assignment 8: Solution

Exercise 1

Theorem (Rice)**.** *A property P of the computable partial functions (c.p.f.) is decidable iff it is* trivial*, i.e., either no c.p.f. has P or all c.p.f. have P.*

The theorem speaks about a property of functions, simply because it is not true for arbitrary properties of programs. For example, the property "the program κ has length of 13 characters" is non-trivial and decidable.

Let P be a decidable and non-trivial property of computable partial functions. We shall give an informal proof of Rice's theorem by reducing the halting problem to the problem of deciding P. Let `P` be any program that decides P, that is for all programs κ we have that

$$\texttt{P}(\kappa) = \begin{cases} \texttt{true} & \text{if } P(\kappa) \\ \texttt{false} & \text{otherwise.} \end{cases}$$

Our goal is to define an algorithm `halts(k, n)` that, given a program κ and an input n, decides whether $\kappa(n)$ halts:

$$\texttt{halts}(\kappa,n) = \begin{cases} \texttt{true} & \text{if } \kappa(n) \text{ halts} \\ \texttt{false} & \text{otherwise.} \end{cases}$$

Now, observe that a never terminating program implements the nowhere defined partial function. Without loss of generality, we can assume that the property P does not hold for all such programs, for otherwise we could choose its complement $\neg P$ instead, which is again decidable and non-trivial. Let `ok` be any program for which P holds. Define `halts` as:

```
def halts(k, n):
    def test(m):
        k(n)
        return ok(m)
    return P(test)
```

If $\kappa(n)$ halts, then `test` behaves like `ok` for all inputs m. If $\kappa(n)$ loops forever, then `test` loops forever for all inputs. Consequently, `halts` uses the algorithm `P` to distinguish between these two cases.

Exercise 2

1. ```
   y = x
   if x < 0:
       y = -x
   assert y >= 0
   ```

   The program states before the assertion $\alpha \iff y \geq 0$ satisfy either one of the constraints

   (a) $\pi_1 \iff x_0 < 0 \wedge x = x_0 \wedge y = -x_0$

   (b) $\pi_2 \iff x_0 \not< 0 \wedge x = x_0 \wedge y = x_0$

   Both $\forall x y x_0 y_0 (\pi_1 \to \alpha)$ and $\forall x y x_0 y_0 (\pi_2 \to \alpha)$ are valid.

2. ```
   if x > y:
       z = x
   else:
       z = y
   assert x <= z and y <= z
   ```

   The program states before the assertion $\alpha \iff x \leq z \wedge y \leq z$ satisfy either one of the constraints:

   (a) $\pi_1 \iff x_0 > y_0 \wedge x = x_0 \wedge y = y_0 \wedge z = x_0$

   (b) $\pi_2 \iff x_0 \not> y_0 \wedge x = x_0 \wedge y = y_0 \wedge z = y_0$

   Both $\forall x y z x_0 y_0 z_0 (\pi_1 \to \alpha)$ and $\forall x y z x_0 y_0 z_0 (\pi_2 \to \alpha)$ are valid.

Exercise 3

Unrolling the loop three times we obtain (`else` branches omitted for clarity):

```
x = 1
y = 1
if x < n:
    assert x == y
    x = x + 1
    y = y + y
    if x < n:
       assert x == y
       x = x + 1
       y = y + y
       if x < n:
           assert x == y
           x = x + 1
           y = y + y
```

The states before the third assertion $\alpha \iff x = y$ satisfy the constraint

$$\pi \iff 1 < n_0 \wedge 2 < n_0 \wedge 3 < n_0 \wedge x = 3 \wedge y = 4 \wedge n = n_0$$

Clearly, the implication $\forall xynn_0(\pi \to \alpha)$ is not valid.

Exercise 4

Suppose we execute the function `main` (see below) concolically with the two symbolic variables b_0 and e_0 for `b` and `e` and that we unroll loops at most twice. For the first concrete execution we assume that `b` and `e` are both `0`.

```
int pow(int b, int e)
{
  int r = b;
  for (int i = 0; i < e; i++)
  {
    r = r * b;
  }
  return r;
}

void main(int b, int e)
{
  var r = pow(b, e);
  if (e % 2 == 0)
  {
    if (r < 0)
    {
      ERROR;
    }
  }
}
```

1. What is the path constraint that will be gathered during this first execution?

   Solution:

   ```
   PC_0 == !(0 < e_0) && (e_0 % 2 == 0) && !(b_0 < 0)
   ```

2. Negate the last conjunct in the path constraint and solve the resulting formula to generate a new input.

   Solution (other solutions are possible): `e_0 == 0, b_0 == -1`

3. What is the path constraint that will be gathered when executing function `main` with the new input?

   Solution (other solutions are possible):

   ```
   PC_1 == !(0 < e_0) && (e_0 % 2 == 0) && (b_0 < 0)
   ```

4. Repeat this process (1. run and record path constraint, 2. negate conjunct in path constraint and generate new input by solving the constraint) until you find an execution that reaches the =ERROR= statement.

   Solution (other solutions are possible): The statement was already reached by the last execution.

5. Compare your concrete inputs to the test cases that are generated by the concolic test-generation tool *Pex* when manually unrolling the loops in the original program. Go to http://www.pexforfun.com/, click on "New", and start from the following program:

```
using System;
using System.Diagnostics.Contracts;

sealed class __DoNotInstrumentAttribute : Attribute { }

// [__DoNotInstrument]
public static class Math
{
    public static int Pow(int b, int e) {
    int r = b;
    for (int i = 0; i < e; i++) {
        r = r * b;
    }
    return r;
    }
}

public class Program
{
    public static int Puzzle(int b, int e) {
    var r = Math.Pow(b, e);
    if (e % 2 == 0) {
        if (r < 0) {
        Contract.Assert(false);
        }
```

```
    }
    return r;
    }
}
```

Solution:

Pex generates the following test case inputs:

(a) `b == 0, e == 0`

(b) `b == int.MinValue, e == 0` (failing)

(c) `b == 0, e == 1`

(d) `b == 0, e == 2`

6. Now, suppose that function `pow` was uninstrumented (e.g., because it was part of a native library). What is the path constraint that will be gathered during the first execution of function =main= (again with `b == 0` and `e == 0`)?

   Solution:

   ```
   PC_0 == (e_0 % 2 == 0)
   ```

7. Negate the last conjunct in the path constraint and solve the resulting formula to generate a new input.

   Solution (other solutions are possible): `e_0 == 1, b_0 == 0`

8. What is the path constraint that will be gathered when executing function `main` with the new input?

   Solution:

   ```
   PC_1 == !(e_0 % 2 == 0)
   ```

9. Is it possible to reach the `ERROR` statement by repeating this process (1. run and record path constraint, 2. negate conjunct in path constraint and generate new input by solving the constraint)?

   Solution: Only if the same constraints are solved multiple times and the solver returns a new solution/input that so happens to reach the `ERROR` statement.

10. Compare your concrete inputs to the test cases that are generated by the concolic test-generation tool *Pex* when manually unrolling the loops in the original program. You can start from the same program as above and you should uncomment the line with the `[__DoNotInstrument]` attribute, which makes all methods in class `Math` uninstrumented.

    Solution: Pex generates the following successful test case inputs, but no failing ones:

(a) `b == 0, e == 0`

(b) `b == 0, e == 1`