Applied Logic Repair Assignment

Mikolaj Hilgert M.H.
m.hilgert@student.fontys.nl
4158385

January 12, 2022

Contents

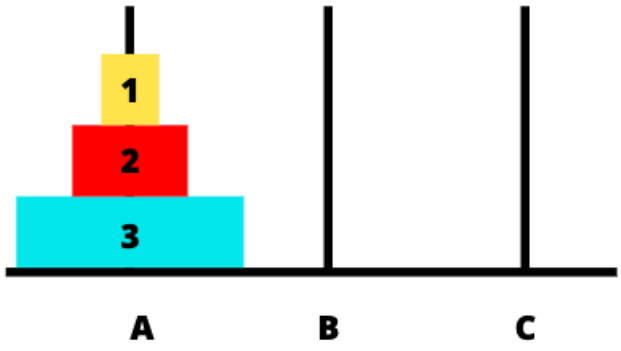
1 Tower's of Hanoi **1**
1.1 Conclusion 6

1 Tower's of Hanoi

The Tower of Hanoi puzzle is a very popular logic puzzle. The point of the puzzle is to move all disks from the first peg, to the third. It consists of three pegs or towers with n disks placed one over the other. It's main rules are the following:

1. No disk can be placed on top of the smaller disk.
2. Disk count and size does not change.
3. Only one disk can be moved at a time.

The instance of the problem I have chosen will be using 3 disks. For simplicity of explanation. The point is to move the Disks from peg A to peg C. Whilst adhering to the aforementioned rules.



A possible approach to this problem with *Z3* is through using a three dimensional board. So that per move there are three pegs and per peg there are three positions. Position 1 will be at the bottom of the peg, Position 2 in the middle and step 3 on the top of the stack. This approach whilst giving a large output, does uphold the integrity of the disk positions. As such an tabular example of step 0 of the start arrange would look as in the following table.

Move	0	0	0
Rod	1	2	3
P3	1	0	0
P2	2	0	0
P1	3	0	0

(Pegs A,B and C will be so fourth referred to as Pegs 1,2 and 3.)

To begin implementation of this puzzle in *Z3*. We define the table/board function, as well as we can define the time point in which *Z3* is to find the solution.

```
;(move,peg,position)->diskVal
(declare-fun B (Int Int Int) Int)

; timepoint at which the required end-state is reached
(declare-const N Int)
```

Now having the board defined function declared, the start and end positions should now be defined. With this approach, we can assert the values we want at any point in time. So, at step 0, for the first rod, the disk at position 1 is 3, at 2 is 2, at 3 is 1. Then, to declare the end-state, we use the N variable and declare that at the third rod, the disk at position 1 is 3, at 2 is 2, at 3 is 1, which is the wanted end state as per the chosen example.

```
(assert (and
    ;start state
    (= (B 0 1 1) 3) (= (B 0 1 2) 2) (= (B 0 1 3) 1)
    (= (B 0 2 1) 0) (= (B 0 2 2) 0) (= (B 0 2 3) 0)
    (= (B 0 3 1) 0) (= (B 0 3 2) 0) (= (B 0 3 3) 0)

    ;wanted-end state
    (= (B N 1 1) 0) (= (B N 1 2) 0) (= (B N 1 3) 0)
    (= (B N 2 1) 0) (= (B N 2 2) 0) (= (B N 2 3) 0)
    (= (B N 3 1) 3) (= (B N 3 2) 2) (= (B N 3 3) 1)
```

Given we have the board, start end states defined we can now describe the first condition. Namely, the one that states that no larger disk may be put on a smaller one. Luckily, with the 3D approach, this is quite trivial. As we are able to assert a forall such that the disk value at position 1 is larger than the value at position 2, and consequently the value at position 2 is to be larger than the one at position 3, at any given step. This simple logic is very efficient as it's easy to extend for larger disk counts. This will be touched upon in a later section.

```
;A disk on the below must be larger or equal to than the
;one above it.
(forall ((t Int) (rod Int))
    (=>
        (<= 0 t N)
        (>= (B t rod 1) (B t rod 2) (B t rod 3))
    )
```

The next condition to be defined, is the one that states that the disk count and size does not change. To define this in *Z3*, we can do so using a combination of two observations. Namely, the fact that across two moves total of disks per move, will not change. As well as, the number of 0's present also in a move should also stay the same. As then we ensure that no new numbers are added, even if they would add up to 6. This is also defined in a forall quantifier with functions.

Function definitions at the top of solutions:

```
; returns the total of all disks and rods per move
(define-fun TotalPerMove((t Int)) Int
        (+
        (B t 1 1) (B t 1 2) (B t 1 3)
        (B t 2 1) (B t 2 2) (B t 2 3)
        (B t 3 1) (B t 3 2) (B t 3 3)
        )
)

; returns the total amount of zeros per move
(define-fun ZeroCountPerMove((t Int)) Int
    (+
        (ite(=(B t 1 1) 0) 1 0) (ite(=(B t 1 2) 0) 1 0)
        (ite(=(B t 1 3) 0) 1 0)
        (ite(=(B t 2 1) 0) 1 0) (ite(=(B t 2 2) 0) 1 0)
        (ite(=(B t 2 3) 0) 1 0)
        (ite(=(B t 3 1) 0) 1 0) (ite(=(B t 3 2) 0) 1 0)
        (ite(=(B t 3 3) 0) 1 0)
    )
)
```

Asserted condition using above defined functions:

```
    (forall ((t Int))
        (=>
            (<= 0 t N)
            (and
                (= (TotalPerMove t) (TotalPerMove (+ t 1)))
                (= (ZeroCountPerMove t) (ZeroCountPerMove (+ t 1)))
            )
        )
    )
```

Next, we must define the last condition, one of disk movement. To do this in *Z3* we make another forall assertion where for every move, there exists three rows. One rod will stay the same (as per the idea that one disk is moved from one rod to another, intrinsically one doesn't change) whilst two change. But the two changing ones have a specific behavior, such that one position in one of the rods becomes 0 (meaning disk was moved form there), and the other one, one position will change into 1, 2 or 3. Whilst the other positions stay the same as in the previous move. This is implemented all in one forall using exists and a function that is used to eliminate redundant code.

Function definitions at the top of solutions:

```
;function for disk movement, that says one position in the rod changes
;and two stay the same. As per rule to move one disk per MOVE.
(define-fun DiskMovement ((t Int) (row Int) (p1 Int) (p2 Int) (p3 Int))
Bool
    (and
        ;different positions
        (distinct p1 p2 p3)
        (<= 1 p1 3)
        (<= 1 p2 3)
        (<= 1 p3 3)

        ;two position on the rod stays the same, whilst one changes
        (distinct (B t row p1) (B (+ t 1) row p1))
        (= (B t row p2)(B (+ t 1) row p2))
        (= (B t row p3)(B (+ t 1) row p3))
    )
)
```

Asserted condition using above defined function. Here the rods are defined and exist is used to assure that this applies to all combinations of rods, and also ensuring, that one rod doesn't change through a move.

```
(forall ((t Int))
    (=>
    ;conditions per move
            (<= 0 t N)
        (exists ((r1 Int) (r2 Int) (r3 Int))
            (and
                ;different rods
                (distinct r1 r2 r3)
                (<= 1 r1 3)
                (<= 1 r2 3)
                (<= 1 r3 3)

                ;One rod doesnt change in any way
                (= (B t r1 3) (B (+ t 1) r1 3))
                (= (B t r1 2) (B (+ t 1) r1 2))
                (= (B t r1 1) (B (+ t 1) r1 1))
```

Continuing, The actual movement of the disk from one rod to another has to be defined. This is done using exists, it is said that there exist two rods in a move where ones position becomes 0, whilst another position on the last rod becomes value 1, 2 or 3. This is why exist is used, so that there is no need to define all possible variations.

```
;There exists a position in a another rod that becomes 0 -> movement
;of the disk
(exists ((p1 Int) (p2 Int) (p3 Int))
    (and
        ;disk movement
        (DiskMovement t r2 p1 p2 p3)
        ;one position becomes 0
        (= (B (+ t 1) r2 p1) 0)
    )
)

;There exists a position in the last rod that becomes either 1,2 or 3.
(exists ((p1 Int) (p2 Int) (p3 Int))
    (and
        ;disk movement
        (DiskMovement t r3 p1 p2 p3)
        ;one position becomes 1, 2 or 3
        (or
            (= (B (+ t 1) r3 p1) 1)
            (= (B (+ t 1) r3 p1) 2)
            (= (B (+ t 1) r3 p1) 3)
        )
    )
)
))))
```

The final thing that has to be added to the large and assert, is the limit on moves to tell *Z3* in how many moves it should be solving in. In this case it is known that it can be done in 7 moves.

```
;limit on moves
(<= 0 N 7)
```

Extension of problem:
To expand this solution into say 4 disks, should not be to difficult in terms of adding any logic, rather exists have to be edited to include the new position/s. But the core logic of comparison of position sizes works due to the >= signs, which means that no 'accepted' combinations have to be defined.

The Output:
Whilst the output is quite lengthy as its per position, and if this solution would be expanded it would become quite a lot larger. It does account for every position and does ensure position integrity of the solution. Counting functions do not have to be updated as they are self defining because of the first manual start input.

1.1 Conclusion

As a result we can see that $Z3$ gives a solution in 7 moves. We can interpret it as a table.

Move	0	0	0
Rod	1	2	3
P3	**1**	**0**	**0**
P2	**2**	**0**	**0**
P1	**3**	**0**	**0**
Move	1	1	1
Rod	1	2	3
P3	**0**	**0**	**0**
P2	**2**	**0**	**0**
P1	**3**	**0**	**1**
Move	2	2	2
Rod	1	2	3
P3	**0**	**0**	**0**
P2	**0**	**0**	**0**
P1	**3**	**2**	**1**
Move	3	3	3
Rod	1	2	3
P3	**0**	**0**	**0**
P2	**0**	**1**	**0**
P1	**3**	**2**	**0**

Move	4	4	4
Rod	1	2	3
P3	**0**	**0**	**0**
P2	**0**	**1**	**0**
P1	**0**	**2**	**3**
Move	5	5	5
Rod	1	2	3
P3	**0**	**0**	**0**
P2	**0**	**0**	**0**
P1	**1**	**2**	**3**
Move	6	6	6
Rod	1	2	3
P3	**0**	**0**	**0**
P2	**0**	**0**	**2**
P1	**1**	**0**	**3**
Move	7	7	7
Rod	1	2	3
P3	**0**	**0**	**1**
P2	**0**	**0**	**2**
P1	**0**	**0**	**3**