Solving Takuzu puzzles

In this project, we are interested in specifying and proving correct a program that solves the so-called Takuzu puzzles (`https://en.wikipedia.org/wiki/Takuzu`). The rules of these puzzles are detailed below.

This project is to be carried out using the Why3 tool (version 1.3.3), in combination with automated provers (Alt-Ergo 2.3, CVC4 1.7 and Z3 4.8). You can use other automatic provers or versions if you want, if they are freely available and recognized by Why3. You may use Coq for discharging particular proof obligations, although the project can be completed without it. The installation procedure may be found on the web page of the course at URL `https://marche.gitlabpages.inria.fr/lecture-deductive-verif/install.html`.

The project must be done individually—team work is not allowed. In order to obtain a grade for the project, you must send an e-mail to `Claude.Marche@inria.fr` and `Jean-Marie.Madiot@inria.fr`, no later than **Thursday, February 25th, 2021** at 22:00 UTC+1. This e-mail should be entitled "MPRI project 2-36-1", be signed with your name, and have as attachment an archive (`zip` or `tar.gz`) storing the following items:

- The canvas file `takuzu.mlw` completed by yours.
- The content of the sub-directory `takuzu` generated by Why3. In particular, this directory should contain session files `why3session.xml` and `why3shapes.gz`, and Coq proof scripts, if any.
- A PDF document named `report.pdf` in which you report on your work. The contents of this report counts for at least half of your grade for the project.

The report must be written in French or English, and should typically consist of 3 to 6 pages. The structure should follow the sections and the questions of the present document. For each question, detail your approach, focusing in particular on the design choices that you made regarding the implementations and specifications. In particular, loop invariants and assertions that you added should be explained in your report: what they mean and how they help to complete the proof.

A typical answer to a question or step would be: *"For this function, I propose the following implementation: [give pseudo-code]. The contract of this function is [give a copy-paste of the contract]. It captures the fact that [rephrase the contract in natural language]. To prove this code correct, I need to add extra annotations [give the loop invariants, etc.] capturing that [rephrase the annotations in english]. This invariant is initially true because [explain]. It is preserved at each iteration because [explain]. The post-condition then follows because [explain]."*

The reader of your report should be convinced at each step that the contracts are the right ones, and should be able to understand why your program is correct, e.g. why a loop invariant is initially true, why it is preserved, and why it suffices to establish the post-condition. It is legitimate to copy-paste parts of your Why3 code in the report, yet you should only copy the most relevant parts, not all of your code. In case you are not able to fully complete a definition or a proof, you should carefully describe which parts are missing and explain the problems that you faced.

In addition, your report should contain a conclusion, providing general feedback about your work: how easy or how hard was it, what were the major difficulties, was there any unexpected result, and any other information that you think is important to consider for the evaluation of the work you did.

1 Description of Takuzu puzzles

Solving a Takuzu puzzle aims at filling a grid with zeros and ones. There exists variations on the size of such a grid, here we fix the size of considered grids to 8×8. An initial configuration of the puzzle is a grid

filled with a few zeros and ones, and the goal is to fill in the remaining empty cells, while respecting the three rules below:

- in any line or column, there should never be three consecutive identical numbers;
- in each line and column, there is the same number of zeros and ones (that is four of each);
- there should be no two identical lines, and no two identical columns

An example of puzzle is as follows

							0
							1
					0		
0							
							0
					1	1	
		1				1	
1			0	0			

and the unique solution of it is

1	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1
1	0	1	1	0	0	1	0
0	1	0	1	1	0	0	1
1	0	1	0	1	1	0	0
0	1	0	1	0	1	1	0
0	0	1	0	1	0	1	1
1	0	1	0	0	1	0	1

2 Appetizers: Basic Functions on Arrays

We start gently with a few basic functions on arrays for checking properties similar to the three different rules above. For this section, we simply consider functions on arrays of integers.

2.1 Check for Consecutive Zeros

As a starter, we consider a question similar to the first Takuzu rule: we want to check whether, in an array of integers, there exists three consecutive zeros. To start with, we want to formalize that expected property as a predicate of the form

```
predicate no_3_consecutive_zeros (a:array int)
```

which is supposed to evaluate to true when there is no three consecutives zeros. To define it, we consider an auxiliary predicate `no_3_consecutive_zeros_sub` taking a bound l and expresses that there is no three consecutive zeros in the sub-array of a between indices 0 (included) and l (excluded).

1. *Fill in the definitions of predicates* `no_3_consecutive_zeros_sub` *and* `no_3_consecutive_zeros` *in the file* `takuzu.mlw`.

We then consider a first implementation, that iterates on the array and checks for each index i if $a[i]$, $a[i+1]$ and $a[i+2]$ are null.

2. *In file* `takuzu.mlw`*, complete the definition of the function* `no_3_consecutive_zeros_version_1` *that implements that first version. Provide the necessary loop invariants to prove this function correct.*

To avoid accessing thrice each array elements, we now consider a second version which records in local variables the last two elements visited.

3. *Complete the definition of the function* `no_3_consecutive_zeros_version_2` *that implements that second version. Provide the necessary loop invariants to prove this function correct.*

A third algorithm is to avoid to record the values of the last two elements visited, but just to record how many zeros where recently visited.

4. *Complete the definition of the function* `no_3_consecutive_zeros_version_3` *that implements that third version. Provide the necessary loop invariants to prove this function correct.*

2.2 Checking for Same Number of Zeros and Ones

We now consider the second Takuzu rule, that requires to have the same number of zeros and ones in each line and column. Again, we simply consider array of integers for now. To start with, we want to define a general logic function that gives the number of occurrences of a given element `f` in an abstract sequence, the latter being represented by a function and two bounds:

```
function num_occ (e:int) (f:int -> int) (i j :int) : int
  (** number of 'l', 'i <= l < j', such that 'f l' is equal to 'e' *)
```

Notice that the first index i is counted but not the last index j. A canvas of its definition is given in the file `takuzu.mlw` under the form of a *ghost recursive function*.

5. *Complete the definition of the function* `num_occ`.
6. *Pose an appropriate post-condition on the auxiliary program function* `count_number_of` *so as to make the other program function* `same_number_of_zeros_and_ones` *provable.*
7. *Prove the auxiliary program function* `count_number_of`.

2.3 Checking for identical sub-arrays

Considering the third Takuzu rule now, we'd like to specify and implement a function that will check if a given array has identical sub-arrays. The expected behaviour is formalized using the following predicate.

```
predicate identical_sub_arrays (a:array int) (o1 o2 l:int)
(** [identical_sub_arrays a o1 o2 l] is true whenever the sub-arrays
    [a[o1..o1+l-1]] and [a[o2..o2+l-1]] are point-wise identical *)
```

8. *Complete the definition of the predicate* `identical_sub_arrays`.
9. *Prove the program function* `check_identical_sub_arrays` *correct by adding appropriate annotations.*

3 Specification of Takuzu puzzles

For representation of Takuzu grids, we choose arrays of size 64 of values of the following type

```
type elem = Empty | Zero | One
type takuzu_grid = array elem
```

The value `Empty` is naturally used to represent partially filled grids. The numbering of cells is as follows.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

To ease the usage of that numbering, the file `takuzu.mlw` introduces a few definitions. The function `column_start_index` gives, for a given cell number, the number of the cell at the top of the same column. The function `row_start_index` gives the number of the cell at the left of the same row.

To denote either a column or a row, we use the term *chunk*. Such a chunk can be represented by a pair of integers $(start, incr)$ where $start$ is the number of the first cell (on the left for rows and at the top for columns) and $incr$ is the increment need for going to the next cell of the chunk (1 for rows and 8 for columns). For simplicity the function `acc` is introduced: `(acc s i k)` denotes the k-th cell of the chunk (s, i) (for $0 \leq k \leq 7$). The predicate `valid_chunk` is also given, to define what are the valid pairs defining chunks.

We now consider each of the three Takuzu rules independently. We are going to write auxiliary functions that check if these rules are valid for a given chunk. Beware that we consider now grids that can be partially filled. That means that from now on, when we say that some rule is valid for some chunk, it ignores the empty cells in the sense that the empty cells of the chunk can still be filled with values for satisfying the given rule.

3.1 The Main Search Algorithm

The main algorithm we want to implement and prove proceeds by a kind of brute-force search, trying to fill in each empty cells with a zero or a one, and checking whether the resulting grid is a solution. Yet, to make such a brute-force algorithm practical, we want to proceed incrementally in the sense that we try to fill empty cells in order, 0 to 63, but at each step we check that the rules are possibly satisfiable before continuing. A pseudo-code for the algorithm is thus as follows

```
solve_aux (g:takuzu_grid) (n:int) (* we assume all cells 0..n-1 already filled *)
= if n=64 then return g (* search completed *) otherwise
  depending in current value of g[n]:
  - if non-empty :
      check if rules are satisfied for cell n, then solve_aux g (n+1)
  - if empty :
      set g[n] to 0, check if rules are satisfied for cell n, and then
        solve_aux g (n+1)
      if the above check failed, or the recursive call to solve_aux did not find a
        set g[n] to 1, check if rules are satisfied for cell n, and then
```

```
      solve_aux g (n+1)
    if the above check failed, or the recursive call did not find any solution :

solve_grid (g:takuzu_grid) = solve_aux g 0
```

The algorithm thus amounts to check whether rules are satisfied but for a given cell only. This is why we start with programming and proving three functions to check Takuzu rules for a given cell, or indeed for a given chunk. A general thing about these functions is that whenever they detect a violation of a rule, then they should raise a common exception `Invalid`.

3.2 First Takuzu Rule for Chunks

Checking the first Takuzu rule for a given chunk is formalized with the following predicate.

```
predicate no_3_consecutive_identical_elem (g:takuzu_grid) (start incr : int) (l:int) =
  (** 'no_3_consecutive_identical_elem g s i l' is true whenever in the
    chunk '(s,i)' of grid 'g', the first 'l' elements do not violate
    the first Takuzu rule *)
```

We remind that this predicate should ignore the empty cells : it should be false only if the given chunk contains three consecutive zero or ones, but not if it contains three consecutive empty cells.

10. In file `takuzu.mlw`, complete the definition of the predicate `no_3_consecutive_identical_elem`.

The program function `check_rule_1_for_chunk` aims at checking Takuzu rule one for a chunk, with an algorithm that ressembles the third algorithm of Section 2.1.

11. Provide appropriate loop invariants to prove this program.

3.3 Second Takuzu Rule for Chunks

We proceed similarly on the second rule: for a given chunk, we want to check that the number of zeros and the number of ones does not exceed 4. The empty cells still do not count, naturally. The specifications and the codes for this part should be inspired from Section 2.2.

12. Complete the ghost function `num_occ` *and the program function* `count_number_of` *that counts the number of a given element in a chunk.*

13. Complete the predicate `rule_2_for_chunk` *and the program function* `check_rule_2_for_chunk` *aiming at checking the Takuzu rule 2 for a given chunk.*

3.4 Third Takuzu Rule for Chunks

For that third rule, when we mean that two chunks are identical, we mean that they have all their cells non-empty and identical. In other words, the check for rule 3 should never raise an invalidity exception when one of the two chunks still contain empty cells.

Moreover, when we say that the rule 3 should be checked for a given chunk, we mean that we should check that this chunk is not identical to any of the other chunks that goes in the same direction (columns or rows).

14. Complete the predicate `identical_chunks` *that must be true when two chunks* (s_1, i) *and* (s_2, i) *are identical on their first* l *elements. Then prove the program* `check_identical_chunks`.

15. Complete the program functions `check_rule_3_for_column` *and* `check_rule_3_for_row`.

Note: once the last two functions are completed, the code for your program is now complete, so you may consider testing your code by running `make tests`. This should run the solver on a few grids, for each of them a solution must be returned, in a fraction of seconds for each of them.

3.5 Checking Rules Satisfaction for a Given Cell

The next step is to code a function that checks whenever, for a given cell number, the Takuzu rules are satisfied for its column and its row. Indeed we are interested in the following predicate.

```
predicate valid_up_to (g:takuzu_grid) (n:int)
(** 'valid_up_to g n' is true whenever all cells with number smaller
    than 'n' satisfy the Takuzu rules *)
```

that states that the rules are satisfied for all cells smaller than the given `n`.

16. *Complete the predicates* `rule_1_for_cell`, `rule_2_for_cell` *and* `rule_3_for_cell` *so has to make the last predicate* `valid_up_to` *mean what it is supposed to mean.*
17. *Complete the program function* `check_at_cell` *that aims at checking* `valid_at_cell g n`*. As for the previous rule checks, that function should raise the exception* `Invalid` *if any rule is violated.*

The next function `check_cell_change` is quite similar to the previous function `check_at_cell`, the only difference, as seen in its body, is that it first sets the content of the cell number `n` to the given element `e`. This function is intended to be used incrementally, that is it assumes the rules were already checked for smaller values of `n`. Notice how the already given pre-condition "`valid_up_to (g[n<-Empty]) n`" expresses that rules were checked for values smaller than `n`, but when assuming the cell `n` is empty.

18. *Complete the specifications of* `check_cell_change` *so as to prove all its verification conditions, except the given assertion and the post-condition "*`valid_up_to g (n+1)`*". Explain why proving the assertion is difficult. Explain also why proving the post-condition, even assuming the assertion valid, is difficult.*
19. *State an appropriate framing lemma before the definition of* `check_cell_change` *so as to prove the assertion that remained unproved in the previous question. Without proving that lemma, explain informally why it is correct.*
20. *State another framing-like lemma before the definition of* `check_cell_change` *so as to prove the post-condition. Without proving that lemma, explain informally why it is correct.*

3.6 Proving the Main Algorithm

The main algorithm is already implemented in the canvas file `takuzu.mlw`. To specify the expected behavior of the solver, we introduce two extra predicates

```
 predicate full_up_to (g:takuzu_grid) (n:int)
(** 'full_up_to g n' is true whenever all the cells lower than 'n' are non-empty *)

predicate extends (g1:takuzu_grid) (g2:takuzu_grid) =
(** 'extends g1 g2' is true when 'g2' is an extension of 'g1', that is all
    non-empty cells of 'g1' are non-empty in 'g2' and with the same value. *)
```

The post-condition of the main algorithm `solve` is thus

```
let solve (g:takuzu_grid) : unit
  ensures { full_up_to g 64 }
  ensures { extends (old g) g }
  ensures { valid_up_to g 64 }
```

meaning that the final value of grid `g` is full, contains the initial grid, and of course satisfy the Takuzu rules.

21. *Prove the main algorithm. You may identify that function* `check_cell_change` *may need some extra post-conditions. You may also identify some extra lemmas to be proved. In your report, try to explain incrementally which verification conditions you solve and how you proved them. In case you add post-conditions to* `check_cell_change`*, you have to prove them valid. The extra lemmas you add may be left unproved, if you carefully explain why you believe they are valid.*

Note that for the question above, it is recommended that you write your report at the same time you do the proof, so has to carefully explain the steps you had to follow.

4 Extra Questions, Discussions, Conclusions

22. *As a bonus, you may prove the lemmas left unproved above. Do not forget to explain how you proved these lemmas.*

23. *Our algorithm is proved correct in a sense that when it returns a grid, it is guaranteed to be a valid one. Another desirable property would be its* completeness*: when the given grid is solvable, our algorithm should return a solution. Without trying to do it in practice, discuss what should be done to extend the current specifications of the code. What would be the main difficulties for proving that completeness property?*

24. *As a conclusion of your report, comment on the difficulties you faced during proving this Takuzu solver program.*