MPRI 2.36.1 Project

Thiago F Cesar

1 Introduction

This report documents the process of proving the correctness of an algorithm for solving Takuzu puzzles.

2 Basic Functions on Arrays

2.1 Check for Consecutive Zeros

1) The predicate no_3_consecutive_zeros_sub looks at windows of length three whose indices ranges between $(i = 0, i + 1 = 1, i + 2 = 2)$ and $(i = l - 3, i + 1 = l - 2, i + 2 = l - 1)$, and thus the first index is given by $0 \leq i < l - 2$. The predicate no_3_consecutive_zeros is derived from the previous one, by looking trough the whole array — that is, with $l = a.length$.

2) For the same reason as explained in the previous question, the index i ranges through $0, ..., a.length - 3$. The proposed invariant $\pi_1 : \forall j.0 \leq j < i \rightarrow \neg\, a[j] = a[j+1] = a[j+2] = 0$ clearly proves the direct implication of the post-condition. Indeed, if the result is true then the loop finished executing and thus with $i = a.length - 2$ we get the definition of no_3_consecutive_zeros. For the reverse implication, it results from the fact that if there are no consecutive zeros then we must finish the loop without an exception.

The invariant is trivially true for $i = 0$. Moreover, if we finish executing an iteration without an exception, then $a[i] = a[i+1] = a[i+2] = 0$ must be false, and by combining with the current invariant $\forall j.0 \leq j < i \rightarrow \neg\, a[j] = a[j+1] = a[j+2] = 0$ we obtain the invariant for the iteration $i + 1$.

3) Differently from the first case, the index i no longer represents the first position of the window, but rather the last: now the window ranges from $(last2 = 0, last1 = 1, i = 2)$ to $(last2 = l - 3, last1 = l - 2, i = l - 1)$. Therefore, the index i ranges through $2, ..., l - 1$.

The first loop invariant $\pi_1 : last1 = a[i - 1] \wedge last2 = a[i - 2]$ expresses the fact that the variables $last1$ and $last2$ store the other two positions on the window. It is clearly true for the initial iteration. Moreover, given that during an iteration the current value of $a[i]$ is put in $last1$ whereas the previous value of $last1$ (which is equal to $a[i - 1]$ by the invariant) is put in $last2$, we can also clearly see that the invariant is preserved. This invariant is used to prove second one.

The second invariant $\pi_2 : \forall j.0 \leq j < i - 2 \rightarrow \neg\, a[j] = a[j + 1] = a[j + 2] = 0$ is virtually identical to the invariant of **2)**, the only difference being that the index i now represents the last position of the window. It is trivially true for the initial case, and if we reach the end of an iteration then the exception was not raised and thus we must have $\neg\, a[i] = last1 = last2 = 0$. By the first invariant this means $\neg\, a[i] = a[i-1] = a[i-2] = 0$, and by combining with $\forall j.0 \leq j < i-2 \rightarrow \neg\, a[j] = a[j+1] = a[j+2] = 0$ we see the invariant also holds for $i + 1$.

Because invariant π_1 assures $last1 = a[i - 1]$ and $last2 = a[i - 2]$, and by the code we have $v = a[i]$, thus assuming the result is $\perp$ then by the code this was a consequence of $v = last1 = last2 = 0$, which means no_3_consecutive_zeros does not hold. Conversely, if the result was true, it must been because we finished executing the for loop, and by the last loop invariant we get the definition of no_3_consecutive_zeros.

4) In this implementation we must go through all of the array to check whether three zeros occur consecutively, and thus the index i should range from 0 to $a.length - 1$.

The first invariant proposed $\pi_1 : 2 \geq count\_zeros \geq 0$ defines the range in which the variable $count\_zeros$ varies, which will be useful to prove the invariant that proves the algorithm's specification. It is true in the first iteration, as $count\_zeros$ is initially 0. Moreover it is preserved at each iteration, as by the induction hypothesis the value at the beginning of the iteration is between 0 and 2, and its value can only increase by one if it is different from 2 (thus staying less then 2), or it can go down to 0. This implies that the final value at the end of the iteration is also between 0 and 2.

The three invariants $\pi_2 : count\_zeros = 0 \rightarrow (i = 0 \vee \neg\ a[i-1] = 0)$, $\pi_3 : count\_zeros = 2 \rightarrow (i \geq 1 \wedge a[i-1] = 0 \wedge (i = 1 \vee \neg\ a[i-2] = 0))$ and $\pi_4 : count\_zeros = 3 \rightarrow (i \geq 2 \wedge a[i-1] = 0 \wedge a[i-2] = 0 \wedge (i = 2 \vee \neg\ a[i-3] = 0))$ allows us to derive conclusions from the value of $count\_zeros$. As $count\_zeros$ equals 0 initially, the invariants π_3 and π_4 are trivially true for the initialization, while π_2 holds because $i = 0$.

Moreover, the invariants are preserved. Indeed, by the invariant π_1 the value of $count\_zeros$ varies in $\{0, 1, 2\}$, and thus it suffices to do a case disjunction on its value at the end of a loop execution.

If $count\_zeros = 0$ then π_3 and π_4 are trivially true and we are left to prove π_2. If $count\_zeros = 0$, then by the algorithm we have seen a non zero element at the previous iteration, that is, $\neg a[i-1] = 0$.

If $count\_zeros = 1$ then π_2 and π_4 are trivially true and we are left to prove π_3. If $count\_zeros = 1$, then the only possibility is that we had $count\_zeros = 0$ on the beginning of the loop and then we added one. As we added one, it follows that $a[i-1] = 0$. Moreover, we have $i \geq 1$ because if we had $i = 0$ then that would imply $count\_zeros = 0$. Finally, as on the beginning of the iteration we had $count\_zeros = 0$ we can conclude by the induction hypothesis of π_2 that this was either the first iteration or we saw an element different from $zero$ on the preceding iteration. Formally, this means $i = 1 \vee \neg\ a[i-2] = 0$.

If $count\_zeros = 2$ then π_2 and π_3 are trivially true and we are left to prove π_4. If $count\_zeros = 2$, then the only possibility is that we had $count\_zeros = 1$ on the beginning of the loop and then we added one, and thus $a[i-1] = 0$. Because we had $count\_zeros = 1$ at the beginning of this previous loop, we apply the invariant π_3 to find $(i - 1 \geq 1 \wedge a[i-2] = 0 \wedge (i = 2 \vee \neg\ a[i-3] = 0))$, showing the rest.

To prove the last invariant $\pi_5 : \forall j.3 \leq j \leq i \rightarrow \neg\ a[j-1] = a[j-2] = a[j-3] = 0$ first note that it is trivially true for the first iteration. The first iteration for each it is not trivially true is the beginning of iteration corresponding to $i = 3$, so in the following we assume $i \geq 3$. Therefore, in this context we can write invariants π_4 as $\pi_2 : count\_zeros = 0 \rightarrow \neg\ a[i-1] = 0$, $\pi_3 : count\_zeros = 2 \rightarrow (a[i-1] = 0 \wedge \neg\ a[i-2] = 0)$ and $\pi_4 : count\_zeros = 3 \rightarrow (a[i-1] = 0 \wedge a[i-2] = 0 \wedge \neg\ a[i-3] = 0)$.

Thus by doing a disjunction of cases on $count\_zeros$ we always get to the conclusion that $a[i-1]$ or $a[i-2]$ or $a[i-3]$ is different from zero, that is, $\neg\ a[i-1] = a[i-2] = a[i-3] = 0$. By combining this with the invariant of the previous iteration, which says $\forall j.3 \leq j \leq i-1 \rightarrow \neg\ a[j-1] = a[j-2] = a[j-3] = 0$, we get the desired result.

To show the reverse implication of the post-condition, first note that if the result was $\perp$ then we must have had $a[i] = 0$ at a point in which $count\_zeros = 2$. By the invariant π_4, we thus also had $a[i-1] = a[i-2] = 0$, showing we saw three consecutive zeros.

Conversely, if the result was true, by the invariant π_5 of the last iteration we have $\forall j.3 \leq j \leq l \rightarrow \neg\, a[j-1] = a[j-2] = a[j-3] = 0$, which one can rewrite to $\forall j.0 \leq j < l-2 \rightarrow \neg\, a[j+2] = a[j+1] = a[j] = 0$ with the substitution $k := j - 3$, showing the result.

2.2 Checking for Same Number of Zeros and Ones

5) It is clear that if $j \leq i$ then the value of num_occ must be zero. If this is not the case we check the value of f at position $j-1$ and we do a recursive call with a smaller j, adding one to the result if the value at position $j-1$ was equal to e. As the value of $j-i$ of the recursive call is positive and decreasing with respect to the recursive calls, this is an appropriate variant.

6) As the function count_number_of calculates the value of num_occ along all the array, for it to be correct we must have $result = num\_occ\ e\ a.elts\ 0\ a.length$. This clearly makes same_number_of_zeros_and_ones provable.

7) In order for the function count_number_of compute the occurrences of e in the array, we must at each iteration have an if to check if $a[i] = e$ and increase n by one it that is the case.

The invariant $n = num\_occ\ e\ a.lets\ 0\ i$ captures the fact that the value of n represents the number of times e was seen before index i. This invariant is trivially true for the initialization phase. Moreover, to see it is preserved, note that on the beginning of the iteration we had $n = num\_occ\ a\ e\ a.elts\ 0\ (i-1)$ and at the end of the loop n equals to $if\ \ a[i] = e\ \ then\ \ (1 + num\_occ\ \ a\ \ e\ \ a.elts\ \ 0\ \ (i-1))\ \ else\ \ (num\_occ\ a\ e\ a.elts\ 0\ (i-1))$. Thus, we find exactly the definition of $num\_occ\ a\ e\ a.elts\ 0\ i$.

By the loop invariant of the last iteration, the result equals $num\_occ\ a\ e\ a.elts\ 0\ a.length$, showing the post-condition.

2.3 Checking for identical sub-arrays

8) According to the specification, identical_sub_arrays should be true whenever $a[o1..(o1+l-1)]$ and $a[o2..(o2+l-1)]$ are point-wise identical. Thus, for $0 \leq j < l$ we must have $a[o1+j] = a[o2+j]$.

9) This function iterates through both parts of the array simultaneously and raises and exception whenever two elements differ. As $o1$ and $o2$ represent the offset on the array, and l represents the length of the subarray, it is clear that they all must be positive. Moreover, in order for the subarrays to be inside a we also must have $0 \leq o1 + l - 1 < a.length$ and $0 \leq o2 + l - 1 < a.length$.

The invariant $identical\_sub\_arrays\ a\ o1\ o2\ k = true$ expresses the fact that if we are still iterating in the for loop then no exception was raised and all the previous elements must have been identical.

To see this invariant holds, first note that for the initialization it holds trivially. It is preserved because, if after an iteration we reach the end of the loop, then no exception was raised and thus we must have $a[o1+k] = a[o2+k]$. By the previous invariant, we have $\forall j.0 \leq j < k \rightarrow a[o1+j] = a[o2+j]$, and coupled with $a[o1+k] = a[o2+k]$ we get $identical\_sub\_arrays\ a\ o1\ o2\ (k+1)$, the invariant of the next iteration

If the function returns $\perp$ then for some k we have $\neg\, a[o1+k] = a[o2+k]$ and thus $identical\_sub\_arrays\ a\ o1\ o2\ l$ must not hold. Conversely, if the function returns true then by the last loop invariant we have $identical\_sub\_arrays\ a\ o1\ o2\ l$.

3 Specification of Takuzu puzzles

3.1 The Main Search Algorithm

No questions.

3.2 First Takuzu Rules for Chunks

10) As done previously, we look at windows of size three with the first corresponding to k, and thus we state a condition for $0 \leq k < l - 2$. The rule is not violated whenever for each k we have that the values of positions k, $k+1$ and $k+2$ are not both mutually equal and equal to either 1 or 0, that is, $\neg(acc\ g\ start\ incr\ k = acc\ g\ start\ incr\ (k+1) = acc\ g\ start\ incr\ (k+2) \wedge (acc\ g\ start\ k\ incr = One \vee ac\ g\ start\ k\ incr = Zero))$. We must have the last to condition to check that the values are different from Empty, as this case does not violate the rule.

11) The proposed loop invariants are almost the same as the ones proposed for question **4)**, with the only difference being that we must do the work two times, to check for zeros and for ones. As question **4)** takes a whole page to explain in details these invariants and why they imply the post-condition, this will not be repeated here.

3.3 Second Takuzu Rule for Chunks

12) For the function num_occ, as in question **5)** we use an if to check if the window is of size zero ($l = 0$), in this case returning trivially zero. If this is not the case, we do a recursive call on the window $0, ..., l-2$ and we add to it one if the value at position $l-1$ equals e. As in the recursive call $l-1$ is positive and smaller then l, it is an adequate variant. The required pre-conditions state that the game board is of right size, that the length of the window is not greater then the one of the board and that we are given a valid chunk.

For the function count_number_of, we repeat almost the same function as the one used in subsection 2.2. We iterate the chunk with a for loop and we add one to n each time we see e. The invariant $n = num\_occ\ e\ g\ start\ incr\ i$ captures the fact that n equals the number of times we have seen e before index i. The same proof as the one given in **7)** shows it is verified, and that it implies the post-condition. The pre-conditions state that the game board is of right size and that we are given a valid chunk.

13) In order for rule two to be followed, we must have less then 4 ones and less then 4 zeros on the chunk. Using the previously defined num_occ, this is expressed as $num\_occ\ Zero\ g\ start\ incr\ 8 \leq 4$ and $num\_occ\ One\ g\ start\ incr\ 8 \leq 4$. This defines predicate rule_2_for_chunk.

Function check_rule_2_for_chunk does the computational equivalent of rule_2_for_chunk. If we have either more then 4 zeros or more then 4 ones, we raise an exception. Otherwise, we can use the post-conditions of the calls to count_number_of to show this function's post-condition. Thus, the function finishes without an exception if and only if rule_2_for_chunk is true, showing the post-condition.

3.4 Third Takuzu Rule for Chunks

14) Predicate identical_ chunks should be true whenever two chunks are complete and equal in their first l positions. That is, given two chunks $(s1, i)$ and $(s2, i)$ and an integer l, we must have for each

position $0 \leq k < l$ on the chunk $acc\ g\ s1\ incr\ k = acc\ g\ s2\ incr\ k \neq Empty$. Equivalently,

$$\forall k.0 \leq k < l \rightarrow (acc\ g\ s1\ incr\ k = acc\ g\ s2\ incr\ k) \wedge \neg(acc\ g\ s2\ incr\ k = Empty)\,.$$

To prove function check_identical_chunks we apply the same strategy as in question 9, but now it is not sufficient to ask the elements to be equal, as we also have to check that cells are non-empty. The invariant

$$\forall k.0 \leq k < i \rightarrow (acc\ g\ s1\ incr\ k = acc\ g\ s2\ incr\ k) \wedge \neg(acc\ g\ s2\ incr\ k = Empty)$$

expresses the fact that, if we get to a certain index i without throwing an exception, then it must be that the first i elements of the arrays are both equal and non-empty. The same proof as the one given in question 9 shows it is correct and implies the post condition. For the pre-conditions, we have as usual that the game board should be of the right size and both chunks should be valid.

15) In order to check that a given column c starting at index $start$ does not equal another one, we can use the previously defined function check_identical_chunks and iterate it, checking equality with all the other columns. As we are dealing with column chunks, then the start indices should be less then 8 and the increments should be of 8, so for $i = 0, .., 7$ we should raised invalid whenever $check\_identical\_chunks\ g\ start\ i\ 8$ and $i \neq start$.

The invariant $\forall k.0 \leq k < i \wedge k \neq start \rightarrow \neg(identical\_columns\ g\ start\ k)$ expresses the fact that, if we have arrived at an index i without throwing an exception, then all the previous columns different from c should be non-full or have some different entry from c. It is trivially initially true. Moreover, it is preserved because if we get to the end of the iteration without throwing an exception then either $i = start$ or $\neg(identical\_columns\ g\ start\ i)$, which composed with the current invariant yields the one for index $i + 1$. This invariant proves the post-condition, as the invariant for the last iteration is equal to the post-condition. In what concerns the pre-condition, it requires the game board to be of the right size and the chunk start index to be valid, given it is of type column (with increment 8).

The case for checking the rows is virtually identical to the previous one, with only minimal changes (for instance, the increment is changed to 1, as we deal now with row chunks).

3.5 Checking Rules Satisfaction for a Given Cell

16) The predicate valid_for_cell should be valid for a certain i when its corresponding row and column satisfy all the rules. The predicates rule_1_for_cell and rule_2_for_cell with n are valid whenever the row and column chunk corresponding to n verifies these rules. Thus, they are simply (and respectively) given by

$$rule\_1\_for\_chunk\ g\ cs\ 8 \wedge rule\_1\_for\_chunk\ g\ rs\ 1$$

and

$$rule\_2\_for\_chunk\ g\ cs\ 8 \wedge rule\_2\_for\_chunk\ g\ rs\ 1\,,$$

where the increments are set to 8 in the case of columns chunks and to 1 in the case of row chunks. For the predicate rule_3_for_cell with n to be true we should have for all $i = 0, .., 7$ that the column chunk starting at i is either the column chunk of n or is not identical to the one of n, and that the row chunk starting at $8 \cdot i$ is either the chunk row of n or is not identical to the one of n. In mathematical terms, it should be true when

$$\forall i.0 \leq i < 8 \rightarrow (i = cs \wedge \neg(identical\_columns\ g\ cs\ i)) \vee (i = rs \wedge \neg(identical\_rows\ g\ rs\ (8 \cdot i)))\,.$$

17) The function check_at_cell is almost complete, we only add the pre-conditions that the game board should be of right size and that the cell's index should be inside the board. If any of the check rule functions that we have already implemented and proved raises an exception, then it will propagate to this function, which will also raise an exception. On the other hand, if all the functions successfully terminate, then we will get all their post-conditions, which directly implies valid_for_cell g n.

18) For the pre-condition, as usual we require the game board to be o the right size and furthermore the cell's index to be inside the game board. This proves that the index n is inside the array's bounds, and thus also that the call to function check_at_cell is a valid one. The exceptional post-conditional is trivially true.

Showing the assertion is non-trivial, because we would need a framing lemma to now that the predicate $valid\_up\_to\ g[n \leftarrow Empty]\ n$ is preserved at the assignment. The same is true for the post-condition, as even though we intuitively know that by adding a correct element at position n we cannot alter the correctness of previous positions, this statement also needs a framing lemma which is non-trivial.

19) To prove the assertion we need the framing lemma

$$\begin{aligned} framing\_assert :& \forall g : takuzu\_grid, n : \mathbb{N}, e_1 : elem, e_2 : elem. \\ & 0 \leq n < g.length \rightarrow g[n \leftarrow e_1][n \leftarrow e_2] = g[n \leftarrow e_2] \end{aligned}$$

which states that if n is an index inside the game board, then first setting $g[n]$ to e_1 and then setting it to e_2 is the same as setting it to e_2 directly. Informally, we can see easily that this is true, given that the attribution of a value to a certain position of an array does not alter any of the other cells. By using this lemma we get $g[n \leftarrow Empty] = g[n \leftarrow e][n \leftarrow Empty]$, and thus $valid\_up\_to\ (g[n \leftarrow Empty])\ n \leftrightarrow valid\_up\_to\ (g[n \leftarrow e][n \leftarrow Empty])\ n$, showing the pre-condition is equivalent to the assertion.

20) To prove the post-condition we need the framing lemma

$$\begin{aligned} framing\_pos :& \forall g : takuzu\_grid, n : \mathbb{N}.\ g.length = 64 \rightarrow 0 \leq n < 64 \\ & valid\_up\_to\ (g[n \leftarrow Empty])\ n \rightarrow valid\_for\_cell\ g\ n \rightarrow valid\_up\_to\ g\ (n+1) \end{aligned}$$

which states that if the game board is of the right size and n is an index inside the game board, then if by setting n to empty in g we get a valid board up to n[1] , and if the cell at position n in g is valid, then the game board is valid up to $n+1$. To see why this lemma is true we assume all its hypothesis and we analyze all the three rules.

1. Rule 1 : If $valid\_for\_cell\ g\ n$ is valid, then the row and the column containing cell n must not have three consecutive non-empty elements, and thus they satisfy rule 1. All the other rows and columns are left unchanged, so if they satisfied rule 1 before they must still satisfy it.
2. Rule 2 : If $valid\_for\_cell\ g\ n$ is valid, then the number of zeros and ones in the row and column containing cell n must be less then 4, and thus they satisfy rule 2. All the other rows and columns are left unchanged, so if they satisfied rule 2 before they must still satisfy it.
3. Rule 3 : If $valid\_for\_cell\ g\ n$ is valid, then the row and the column containing cell n must be either non-full or different from the other rows and columns, respectively. Thus they satisfy rule 3. All the other rows (and columns) are left unchanged, so if they were mutually different

[1] Remember that valid up to n means the first n positions starting from 0, so it does not include n.

before, and as they are different from the row (and column) containing cell n, then the must stay mutually different.

21) In order to prove the main algorithm, we first note that function solve is already proven true by the pre and post-conditions of function solve_aux, and thus we are left to prove the latter.

We start with the new pre-conditions we add, which are that the game board is of right size and that n should be in $0, .., 64$. Note that, even thought in all previous functions we asked n to be in $0, .., 63$ – as 64 would be out of the game board – here the condition $n = 64$ means the algorithm has finished, and thus this must be included in the pre-condition. We also remark that those new pre-conditions are implied by the ones of solve, so the function solve is still automatically proven correct.

For the variant, we claim we can use $64 - n$. Indeed, as the recursive calls are all done with a greater n, and its value is upper bounded by 64, then $n - 64$ is both non-negative and decreasing through the recursive calls. The new post-condition $g = old\ g$ will be explained in the following.

We now proceed to prove the correctness of the code. First note that for the case $n = 64$ the function returns immediately with the exception SolutionFound. Note that in this case the exception post-conditions are obviously verified. Indeed, we have $full\_up\_to\ g\ 64$ and $valid\_up\_to\ g\ 64$ from the pre-conditions and also $g = old\ g$ because we did not alter the state of g.

Now we are restricted to case $0 \leq n < 64$. We now proceed to prove that the function calls in case Empty are correct. Of all the pre-conditions of function check_cell_change, the only one that is not implied by the pre-conditions of solve_aux is $valid\_up\_to\ g[n \leftarrow Empty]\ n$.

As we have $valid\_up\_to\ g\ n$ by the pre-condition of function solve_aux, we can see intuitively that this implies $valid\_up\_to\ g[n \leftarrow Empty]\ n$. This cannot be deduced by the automatic solvers, so we add a lemma

$$valid\_preserved : \forall g : takuzu\_grid, n : int.\ 0 \leq n < 64 \rightarrow valid\_up\_to\, n \rightarrow valid\_up\_to\, g[n \leftarrow Empty]\, n\,.$$

The reason this lemma must be true is because by setting a cell to empty in a game board that was valid until position n we cannot create a violation of any of the rules before this position.

The call to function check_at_cell can either raise Invalid, and then we jump to a second call to check_at_cell, or can return normally. Assuming it returns normally, we then have a recursive call $solve\_aux\ g\ (n + 1)$, whose pre-conditions we have to prove. The pre-condition $g.length = 64$ is obviously true, whereas the pre-condition $0 \leq n + 1 \leq 64$ is proven from the fact that this part of the code is accessible for only $0 \leq n < 64$. The pre-condition $valid\_up\_to\ g\ (n + 1)$ is given by the post-condition of the call to function check_at_cell.

The pre-condition $full\_up\_to\ g\ (n + 1)$ is not proven by default, and to prove it we need to add to function check_cell_change the post-condition $(old\ g)[n \leftarrow e] = g$ — which is obviously true because it describes the change of the state of g in function check_at_cell. Using this new post-condition, we are able to link the state of g before and after the function call, so we can use the pre-condition $full\_up\_to\ g\ n$, which we had at the beginning of the call of solve_aux, to prove $full\_up\_to\ g\ (n+1)$. Indeed, as $(old\ g)[n \leftarrow Zero] = g$, and $old\ g$ is full up to n, we get that g is full up to $n + 1$.

The recursive call to solve_aux can either raise an exception SolutionFound or return normally. If it raises the exception then as there is no catch for an exception SolutionFound in the body of the function, then this solution will be propagated and the current function call will end. Thus we are left to prove the exceptional post-condition of the current call.

Both $full\_up\_to\ g\ 64$ and $valid\_up\_to\ g\ 64$ are trivially implied from the exceptional post-condition of the recursive call. To show $extends\ (old\ g)\ g$ we first note that the recursive call to solve_aux is done with $(old\ g)[n \leftarrow Zero]$, which extends $old\ g$. By the exceptional post-condition of the recursive call, the value of g after its execution will extend $(old\ g)[n \leftarrow Zero]$, and thus by transitivity we get the result.

Next we have another call to function check_cell_change. We must prove its pre-conditions both in the cases where the first call to check_cell_change raised Invalid and we moved directly to its second call, and also in the case where the first recursive call to solve_aux returned without raising an exception.

For the first case we need again to change check_cell_change, in order to describe how the first call to check_cell_change (which returned with an exception) altered g. We therefore change its exceptional post condition to $Invalid \rightarrow g[n \leftarrow e] = old\ g$, which is obviously true. This allows us once again to link the current state of g to its state on the beginning of solve_aux, and thus we can repeat an analogous proof to show the pre-conditions of the second call to check_cell_change are satisfied.

For the second case, this is initially impossible because the recursive call to solve_aux does not give us any post-conditions to relate the state of g before and after the call. It is thus important to observe that if we make a recursive call and it returns without raising SolutionFound it will modify g in the process but at the end its initial state will be restored. This justifies adding to solve_aux the post-condition $g = old\ g$. Using this post-condition, we can once again relate the current state of g to the one before the recursive call and then give an analogous proof as before to why the pre-conditions of check_cell_change must be satisfied.

Assuming this second call to function check_cell_change does not raise an exception, we need to prove the pre-conditions of the subsequent recursive call to solve_aux. The proof is the same as the one already done and we do not repeat it.

If the second recursive call to solve_aux returns with an exception we must once again show the exceptional post-condition of the current call. Once again, both $full\_up\_to\ g\ 64$ and $valid\_up\_to\ g\ 64$ are trivially implied from the exceptional post-condition of the recursive call. Moreover, the state of g before the second recursive call is described by $(old\ g)[n \leftarrow Zero][n \leftarrow One]$, and thus extends the original state at the beginning of the function. Because the recursive call extends $(old\ g)[n \leftarrow Zero][n \leftarrow One]$, by transitivity we get that $extends\ (old\ g)\ g$ holds.

If the second recursive call to solve_aux returns normally, we are left to prove that the post-condition $g = old\ g$ holds. We first note that, by composing the added post-conditions to check_cell_change and using the fact that the recursive calls do not change g, we obtain that g equals to $(old\ g)[n \leftarrow Zero][n \leftarrow One][n \leftarrow Empty]$ at the end of the code. We thus can see that $old\ g = g$, because we are in the case $g[n] = Empty$. However the solvers will not prove this automatically, so we need to add the lemma

$$\begin{aligned} framing\_last : &\forall g : takuzu\_grid, n : int, e_1 : elem, e_2 : elem, e_3 : elem. \\ &g.length = 64 \rightarrow 0 \leq n < 64 \rightarrow g[n] = e_1 \rightarrow g[n \leftarrow e_2][n \leftarrow e_3][n \leftarrow e_1] = g\,, \end{aligned}$$

which allows us to prove $g = old\ g$.

We have now finished the case Empty and now we proceed to the case Zero/One. The pre-conditions of check_at_cell are all implied by the pre-conditions of solve_aux. If this call raised an exception we finish the code with $g = old\ g$ trivially, because we did not change g. Assuming this call does not raise

an exception, we then proceed to the recursive call to solve_aux. $g.length = 64$, $0 \leq n \leq 64$ and $full\_up\_to\ g\ (n+1)$ are obviously true, and $valid\_up\_to\ g\ (n+1)$ is true because we have both $valid\_up\_to\ g\ n$ and $valid\_for\_cell\ g\ n$.

If the recursive call returns without raising an exception we then get that it did not alter the state of g. Because the code does not alter the state of g before this call, we get $g = old\ g$. If the recursive call returns with an exception, the exceptional post-condition of the recursive call implies the exceptional post-condition of the current call trivially.

4 Extra Questions

23) Because we implement a brute force algorithm, informally it is clear that it is complete, in the sense that if there is a solution it will eventually try it (if it does not return another one before). To show this formally we would first need a predicate $no\_solution\ g$ which should state that all game boards that are full and extend g are invalid. The function solve_aux should have it added to its post-condition in order to express that there are no solutions.

Then we would need to modify the contract of the functions that check for the rules and change their exceptional post-conditions in order for them to ensure that the game board is invalid in the case we find a violation of a rule.

The same should be done with functions check_at_cell and check_cell_change. More precisely, on the case Zero/One of the match in solve_aux, if check_at_cell returns invalid we should get a post-condition saying that all full extensions of g are invalid. Likewise, if on the Empty case of the match we get failures from both calls to check_cell_change, then we should obtain that both $g[n \leftarrow Zero]$ and $g[n \leftarrow One]$ are invalid boards and we should be able to deduce that all full extensions of g are invalid.

An important point is also that when we do a call of the algorithm on a board g and we reach a board g' that extends g and is invalid, the algorithm will not explore the boards that extend g'. Therefore, when our algorithm returns that g' is invalid we should also have a lemma to state that any full extension of g' should be also invalid. Even though we can informally see that this is valid, the proof may be non-trivial due to framing issues.

5 Conclusion

In this project I successfully proved all the functions that compose the takuzu solving algorithm. The steps were explained in detail and all the lemmas left unproven were shown to hold in this document. I will then take the rest of this conclusion to make personal comments on the difficulties that I faced throughout the process.

Firstly, all the questions up to 20 were relatively easy. This part of the project was by far the fastest to do, and the only function that took me some more time was no_3_consecutive_zeros_version_3, to which I had to add a consider amount of loop invariants in order for me to be able to prove it. Those invariants were then readily generalized to prove check_rule_1_for_chunk, which would have been much harder to prove if I had not proved no_3_consecutive_zeros_version_3 before. The questions 18 to 20, in which I had to prove check_cell_change, were also a bit harder.

The question 21 was the one that took me the most time, as I ended up trying various contracts and lemmas before finding the right ones. In retrospective, the final solution is relatively simple, so maybe I

was just no looking at the right direction.

The part of this project which took at least $\frac{2}{3}$ of my time was writing this report. Indeed, the project specification demanded for every question to be explained in detail[2]. Therefore, I found impossible to follow this specification while staying with the usual 3 to 6 page count it also mentioned. Thus, I found best to explain everything with the required level of detail, even thought that meant exceeding the usual page count.

Nevertheless, in order for the report to not exceed too much the usual page count, and also of course to save time, I tried to not repeat myself with proofs that were the same. For example, the contracts that prove the functions that check the first and second rules (question 11 and question 12) are almost the same as the ones seen on section 2, and therefore I did not repeat the proofs.

As a suggestion, I think that it would have been better to have more questions of coding, or maybe harder ones, but in compensation to replace the report with comments in the code. Indeed, most of the proofs are tedious to write, and as each contract of each function needs a proof, the size of the report ends up being huge.

[2] As stated in the 5th paragraph of the first page, beginning with "A typical answer to a question or step would be: For this function, I propose..."