# Rapport du projet: résolution de Takuzu

M2 Mathématiques - Spécialité LMFI

Année universitaire 2020-2021

Université de Paris

Étudiant : Qi QIU    $N^o$ étudiant : 71607226
Enseignant: Claude Marché
Enseignant: Jean-Marie Madiot

# Contents

**1 Appetizers : Basic Fonctions on Arrays** **1**
1.1 Check for Consecutive Zeros . . . 1
1.2 Checking for Same Number of Zeros and Ones . . . 1
1.3 Checking for identical sub-arrays . . . 2
1.4 Second Takuzu Rule for Chunks . . . 3
1.5 Third Takuzu Rule for Chunks . . . 4
1.6 Checking Rules Satisfaction for a Given Cell . . . 5
1.7 Proving the Main Algorithm . . . 6

# 1 Appetizers : Basic Fonctions on Arrays

## 1.1 Check for Consecutive Zeros

**Question 1**
predicate no-3-consecutive-zeros-sub (a:array int) (l:int) = $\forall i.\ 0 \leq i \leq l-3 \rightarrow \neg(a[i] = a[i+1] = a[i+2] = 0)$
predicate no-3-consecutive-zeros-sub (a:array int) (l:int) = no-3-consecutive-zeros-sub a a.length

**Question 2** Dans la première implémentation, l'indice $i$ de la boucle $for$ incrémente de 0 à $a.length - 3$. À chaque itération $i$, la fonction du programme examine si $a[i] = a[i+1] = a[i+2] = 0$ et relève une exception si c'est le cas. Pour prouver cette fonction, j'ajoute un invariant dans la boucle:

```
invariant { no_3_consecutive_zeros_sub a (i+2) }
```

qui capture le fait que, il n'y pas de 3 occurrences consécutives de 0 dans le sous tableau de longueur $(i+2)$ suivant : $a[0], a[1], ..., a[i+1]$. Cette condition est évidement vrai quand $i = 0$ puisque le sous tableau $a[0], a[i]$ ne contient que deux éléments. L'invariant est préservé à chaque itération si la fonction ne relève pas d'exception. Finalement, à la fin du boucle, le programme assure qu'il n'existe pas de 3 occurrences consécutives de 0 dans le sous tableau $a[0], a[1], ..., a[(i-3)+2]$ qui est exactement le tableau considéré. La post-condition s'ensuit.

**Question 3** Dans la deuxième implémentation,l'indice $i$ de la boucle $for$ incrémente de 2 à $a.length-1$. Comme les deux variables $last1$ et $last2$ ne stockent que les valeurs et perdent donc les autres informations. Pour prouver la correction de cette fonction du programme, j'ajoute deux invariants:

```
invariant{ last2 = a[i-2] /\ last1 = a[i-1]}
invariant{ no_3_consecutive_zeros_sub a i }
```

Le premier invariant capture le fait que, à chaque itération $i$, en examinant si $v = last1 = last2 = 0$, la programme vérifie en fait $a[i] = a[i-1] = a[i-2] = 0\ i$. Le premier invariant qui n'est qu'une simple traduction de l'implémentation, est évidement correct. Le deuxième invariant est vrai initialement car un sous tableau de deux éléments ne peuvent pas avoir 3 occurrences consécutives de 0. À chaque itération, cet invariant est préservé si la fonction ne relève pas d'exception. À la sortie de la boucle, on obtient bien la post-condition.

**Question 4** Dans la troisième implémentation, l'indice $i$ de la boucle $for$ incrémente de 0 à $a.length-1$. La version 3 utilise une variable mutable "count-zero" pour distinguer les différents possibilités du sous tableau $a[i-2]a[i-1]a[i]$. c'est incompréhensible pour les solveurs. Ainsi j'ajoute des invariants supplémentaires pour expliquer, aux solveurs, le lien entre les valeurs de la variable et de différents cas possibles.

```
invariant { no_3_consecutive_zeros_sub a i }
invariant { 0 <= count_zeros <= 2 }
invariant { i = 0 -> count_zeros = 0 }
invariant { i = 1 -> count_zeros <= 1 }
invariant { (i = 1 /\ count_zeros = 1) -> a[0] = 0 }
invariant { (i >= 1 /\ count_zeros = 0) -> (a[i-1] <> 0) }
invariant { (i >= 2 /\ count_zeros = 1) -> (a[i-1] = 0 /\ a[i-2] <> 0) }
invariant { (count_zeros = 2 -> (a[i-2] = a[i-1] = 0)) }
```

Le premier invariant est initialement vrai, car le sous tableau est de longueur $-1$. Cet invariant est préservé à chaque itération si une exception n'est pas relevée. À la sortie de la boucle, on obtient bien la post-condition désirée. Les autres invariants ne sont qu'une traduction directe du lien entre les valeurs de la variable $count-zeros$ et les occurrences possibles.

## 1.2 Checking for Same Number of Zeros and Ones

**Question 5** Pour la fonction logique $num\_occ$, je propose l'implémentation suivant.

```
let rec ghost function num_occ (e:int) (f:int -> int) (i j :int) : int
  variant { j - i }
  = if i >= j then 0 else
    if f (j-1) = e then 1 + num_occ e f i (j-1) else num_occ e f i (j-1)
```

Cette fonction examine récursivement les valeurs $f(j-1), f(j-2), ..., f(i)$ pour compter le nombre d'occurrence de $e$. Pour prouver la terminaison de la récurrence, je propose le variant $j-i$.

**Question 6 et 7** Pour la fonction du programme "count-number-of", je propose l'implémentation suivante

```
let count_number_of (e:int) (a:array int) : int
  ensures { result = num_occ e a.elts 0 a.length }
  =
  let ref n = 0 in
  for i=0 to a.length - 1 do
    invariant { n = num_occ e a.elts 0 i }
    if a[i] = e then n <- n + 1
  done;
  n
```

La post-condition de cette fonction capture le fait que le résultat retourné par l'exécution est un entier naturel qui est égal au nombre d'occurrences de $e$ dans le tableau $a$. Afin de prouver la correction du programme, j'ajoute un invariant pour la boucle $for$. Cet invariant capture le fait que à chaque itération $i$, n est égal au nombre d'occurrence de $e$ dans le sous tableau $a[0], a[1], ..., a[i-1]$. Ce invariant est initialement vrai puisque le sous tableau est de longueur 0. Il est préservé à chaque itération $i$ car n est incrémenté de 1 si et seulement si $a[i] = e$. La post-condition s'ensuit à la fin du boucle $for$.

## 1.3 Checking for identical sub-arrays

**Question 8** Pour le prédicat $identical - sub - arrays$, je propose l'implémentation suivante:

```
    predicate identical_sub_arrays (a:array int) (o1 o2 l:int)
= forall k:int. 0 <= k <= (l-1) -> a[o1+k] = a[o2+k]
```

**Question 9** Pour la fonction du programme $check - identical - sub - arrays$, j'ajoute les pré-conditions suivante:

```
requires { 0 <= o1 < a.length /\ 0 <= o2 < a.length }
requires { 1 <= l < a.length }
requires { 0 <= o1 + (l - 1) < a.length /\ 0 <= o2 + (l - 1) < a.length }
```

qui capture les faits que pour tout $i$ varie entre 0 et $(l-1)$, $o1 + (l-1)$ et $o2 + (l-1)$ sont deux indices valides. Afin de prouver la correction du programme, j'ajoute, dans la boucle $for$, l'invariant suivant:

```
invariant { identical_sub_arrays a o1 o2 k }
```

qui capture le fait que, au début de chaque itération $k$, $a[o1], a[o1+1], ..., a[o1+(k-1)]$ et $a[o2], a[o2+1], ..., a[o2+(k-1)]$ sont identique point par point. Cet invariant est initialement vérifié puisque les deux sous tableaux sont de longueur 0. Il est préservé à chaque itération $k$ parce que si le programme ne relève pas d'exception alors $a[o1+k]$ et $a[o2+k]$ sont identiques. Et la post-condition s'ensuit parce que, à la fin du boucle, la propriété " $a[o1], a[o1+1], ..., a[o1+(l-1)]$ et $a[o2], a[o2+1], ..., a[o2+(l-1)]$ " est vérifiée.

**Question 10** Pour le prédicat $no - 3 - consecutive - identical - elem$, je propose l'implémentation suivante:

```
predicate no_3_consecutive_identical_elem (g:takuzu_grid) (start incr : int) (l:int) =
forall k:int. 0 <= k <= l-3 ->
      ( not (g[start+incr*k] = g[start+incr*(k+1)] = g[start+incr*(k+2)]=One)
   /\   not (g[start+incr*k] = g[start+incr*(k+1)] = g[start+incr*(k+2)]=Zero) )
```

**Question 11** Pour capturer le fait que cette fonction de programme ne modifie pas le tableau g, j'ajoute les post-conditions:

```
ensures { g = old g }
raise { Invalid -> g = old g }
```

Et pour prouver la fonction du programme $check - rule - 1 - for - chunk$, dans la boucle du programme, j'ajoute le invariant suivant:

```
invariant { no_3_consecutive_identical_elem g start incr i }
invariant { (i = 0 -> count_zeros = 0) /\ (i = 0 -> count_ones = 0)}
invariant { i = 1 -> (count_zeros <= 1 /\ count_ones <= 1)}
invariant { 0 <= count_zeros <= 2 /\ 0 <= count_ones <= 2}
invariant { (i = 1 /\ count_zeros = 1) -> (acc g start incr 0) = Zero}
invariant { (i = 1 /\ count_ones = 1) -> (acc g start incr 0) = One}
invariant { (i >= 1 /\ count_zeros = 0) -> ( acc g start incr (i-1) ) <> Zero}
invariant { (i >= 1 /\ count_ones = 0) -> ( acc g start incr (i-1) ) <> One}
invariant { (i >= 2 /\ count_zeros = 1) -> (acc g start incr (i-1)) = Zero /\ (acc g start in
invariant { (i >= 2 /\ count_ones = 1) -> (acc g start incr (i-1)) = One /\ (acc g start incr
invariant { i <= 1 \/ (count_zeros = 2 -> (acc g start incr (i-2)) = (acc g start incr (i-1))
invariant { i <= 1 \/ (count_ones = 2 -> (acc g start incr (i-2)) = (acc g start incr (i-1))
```

Le premier invariant capture le fait que au début de l'itération $i$ il n'y a pas de 3 occurrences consécutives de *Zero* ou *One* dans les $l$ premiers éléments de la ligne ou la colonne considérée. Cet invariant est initialement vrai car si $i = 0$, il n'y a pas d'élément. Et l'invariant est préservé à chaque itération si le programme ne relève pas d'exception. À la sortie de la boucle, la post-condition du programme s'ensuit. Comme la troisième implémentation de l'algorithme de la section 2.1, les autres invariants lient les différentes valeurs de la variable *count_zeros* et *count_ones* avec les différentes occurrences possibles.

## 1.4 Second Takuzu Rule for Chunks

**Question 12** Pour la fonction logique $num_occ$, je propose l'implémentation suivante :

```
let rec function num_occ (e:elem) (g:takuzu_grid) (start incr:int) (l:int)
  requires { g.length = 64 /\ valid_chunk start incr /\ 0 <= l <= 8 }
  variant { l }
= if l = 0 then 0
            else match eq (acc g start incr (l-1)) e with
                 |True -> 1 + (num_occ e g start incr (l-1))
                 |False -> num_occ e g start incr (l-1)
                 end
```

La pré-condition capture le fait que le tableau est de taille 64, que $(start, incr)$ décrit bien un chunk et que l ne dépasse le nombre d'éléments d'un ligne (ou une colonne). Le programme compte récursivement le nombre d'occurrence de l'élément $e$ dans les $l$ premiers élément considérés de la ligne (ou colonne). Afin de prouver ce programme récursive, j'ajoute le variant $l$.

Pour prouver la fonction du programme $count - number - of$, je propose la pré-condition:

```
requires { g.length = 64 /\ valid_chunk start incr }
```

et l'invariant de la boucle:

```
invariant { n = num_occ e g start incr i }
```

La précondition capture le fait que le tableau est de taille 64, que $(start, incr)$ décrit bien un chunk. L'invariant capture le fait qu'au début de chaque itération $i$, n est égal qu'au nombre d'occurrences de e dans les i premiers éléments du chunk décrit pas (start,incr). Cet invariant est initialement évidement vrai. Cet invariant est préservé à chaque itération, car n est incrémenté de 1 si et seulement si le $(i + 1) - ime$ élément du chunk considéré est egal à e. La post-condition s'ensuit à la sortie de boucle.

**Question 13** Pour le prédicat $rule - 2 - for - chunk$, je propose l'implémentation suivante:

```
predicate rule_2_for_chunk (g:takuzu_grid) (start incr:int) =
  num_occ Zero g start incr 8 <= 4 /\
  num_occ One g start incr 8 <= 4
```

Et pour la fonction $check - rule - 2 - for - chunk$, je propose l'implémentation suivante:

```
let check_rule_2_for_chunk (g:takuzu_grid) start incr : unit
  =
  if count_number_of Zero g start incr > 4 then raise Invalid;
  if count_number_of One g start incr > 4 then raise Invalid;
```

Le contrat de cette fonction est :

```
requires { g.length = 64 /\ valid_chunk start incr }
ensures { g = old g }
raise { Invalid -> g = old g }
```

La précondition capture le fait que cette fonction de programme ne modifie pas le tableau $g$ et que dans le chunk $(start, incr)$, il y a au plus 4 *Zero* et au plus 4 *One*. La précondition capture le fait que le tableau est de taille 64, que $(start, incr)$ décrit bien un chunk.

### 1.5 Third Takuzu Rule for Chunks

**Question 14** Pour le prédicat, je propose l'implémentation:

```
predicate identical_chunks (g:takuzu_grid) (s1 s2:int) (incr:int) (l:int)
= forall k. 0 <= k < l ->
    (acc g s1 incr k) = (acc g s2 incr k) /\ (acc g s1 incr k) <> Empty
```

Pour prouver la fonction du programme $check - identical - chunks$, je propose l'implémentation suivante

```
let check_identical_chunks g start1 start2 incr : bool
= try
    for i=0 to 7 do
      match acc g start1 incr i, acc g start2 incr i with
      | Zero,Zero -> ()
      | One,One -> ()
      | _ -> raise DiffFound
      end
    done;
    True
  with DiffFound -> False
  end
```

Le contrat de cette fonction est

```
requires { g.length = 64 /\ valid_chunk start1 incr /\ valid_chunk start2 incr }
ensures { result = True <-> identical_chunks g start1 start2 incr 8 }
```

La pré-condition capture le fait que le tableau est de taille 64, que $(start1, incr)$ et $(start1, incr)$ décrit bien deux lignes ou deux colonnes. Les deux post-conditions capture le fait que la fonction logique retourne $True$ si et seulement si les deux chunks sont identiques point par point et tous les cases no-Empty.
L'invariant de la boucle est

```
invariant { identical_chunks g start1 start2 incr i }
```

Il capture le fait qu'au début de chaque itération $i$, les $i$ premiers éléments des chunks $(start1, incr)$ et $(start1, incr)$ identiques point par point et tous non-Empty. Cet invariant est initialement vrai pour $i = 0$. Cet invariant est préservé à chaque itération si la fonction ne relève pas d'exception. À la sortie, on obtient bien la post-conditions: les 8 éléments des deux chunks sont identiques point par point et tous non-Empty.

**Question 15** Pour le prédicat $identical - rows$, je propose l'implémentation:

```
predicate identical_columns (g:takuzu_grid) (s1 s2:int) =
identical_chunks g s1 s2 8 8
```

Pour la fonctions du programme $check\_rule\_3\_for\_row$, je propose l'implémentation, le contrat et l'annotation suivantes:

```
let check_rule_3_for_row (g:takuzu_grid) (start:int) : unit
=
  for i=0 to 63 do
      invariant { forall k. 0 <= k < i /\ (mod k 8 = 0) /\ k <> start ->
                        not (identical_rows g start k) }
      if mod i 8 = 0 then
        if (i <> start) then
          match check_identical_chunks g start i 1 with
          |True -> raise Invalid;
          |False -> ()
          end
  done
```

Le contrat de cette fonction est

```
requires { g.length = 64 /\ 0 <= start < 64 /\ mod start 8 = 0 }
ensures { forall k. 0 <= k < 8 /\ 8*k <> start -> not (identical_rows g start (8*k)) }
ensures { g = old g }
raises { Invalid -> g = old g}
```

La pré-condition capture le fait que le tableau est de taille 64 et que *start* est la première case d'une ligne. Les post-conditions capture les faits que les autres lignes sont différentes de cette ligne et que cette fonction de programme ne modifie pas $g$. Le prédicat $identical-columns$ et la fonction du programme $check\_rule\_3\_for\_column$ sont analogues.

## 1.6 Checking Rules Satisfaction for a Given Cell

**Question 16** les implémentations:

```
predicate rule_1_for_cell (g:takuzu_grid) (n:int) =
  let cs = column_start_index n in
  let rs = row_start_index n in
  rule_1_for_chunk g cs 8 /\ rule_1_for_chunk g rs 1

predicate rule_2_for_cell (g:takuzu_grid) (n:int) =
  let cs = column_start_index n in
  let rs = row_start_index n in
  rule_2_for_chunk g cs 8 /\ rule_2_for_chunk g rs 1

predicate rule_3_for_cell (g:takuzu_grid) (n:int) =
  let cs = column_start_index n in
  let rs = row_start_index n in
  forall i. 0 <= i < 8 ->
        ((i <> cs -> not identical_columns g cs i) /\
         (8*i <> rs -> not (identical_rows g rs (8*i))))
```

**Question 17** Pour prouver la fonction du programme $check-at-cell$, je propose le contrat:

```
requires { g.length = 64 /\ 0 <= n < 64 }
ensures { valid_for_cell g n }
ensures { g = old g }
raise { Invalid -> g = old g }
```

**Question 18** Pour la fonction $check-cell-change$, je propose le contrat:

```
requires { g.length = 64 /\ 0 <= n < 64 }
requires { valid_up_to g[n<-Empty] n }
writes { g }
```

```
    ensures { valid_up_to g (n+1) }
    ensures { g = (old g)[n<-Empty] }
    raises { Invalid -> g = (old g)[n<-e] }
```

**Question 19** Pour prouver l'assertion en question, j'ajoute le lemme suivant:

```
lemma subst_0 : forall g:takuzu_grid, n:int, e:elem.
                0 <= n < g.length -> g[n<-e][n<-Empty] = g[n<-Empty]
```

qui exprime le fait que en substituant la valeur de la case $n$ par $e$ et puis par *Empty*, on obtient le même tableau qu'en substituant directement la valeurs de la case $n$ par *Empty*.

**Question 20** Pour prouver la post-condition, j'ajoute 3 autres lemmes. Le premier lemme "aux-rule-1-and-rule-2-for-chunk" exprime le fait que si une ligne (resp. une colonne) de $g[n \leftarrow Empty]$ différente de celle de la case $n$ satisfait la règles 1 ou la règle 2, alors la même ligne (resp. colonne) de g satisfait la même règle, puisque les contenus des ces case n'ont pas changé après la substitution. Le deuxième lemme "sym" exprime les relations $identical - rows$ et $identical_columns$ sont symétriques. Le troisième lemme "inv-not-identical" exprime le fait que si deux lignes (resp. deux colonnes) de $g[n \leftarrow Empty]$ différentes de celle de la case $n$ sont identiques alors les deux lignes (resp. les deux colonnes) de $g$ sont identiques, puisque les contenus des cases n'ont pas changé après la substitution.

Comme cette fonction est un peu difficile à prouver, afin d'accélérer la démonstration. j'ajoute aussi quelques assertions. Ils sont, la moitié, les assertions pour rappeler les faits utiles qui découlent directement des pré-conditions ou l'assertion mentionnée dans la question 19, par exemple l'assertion suivante:

```
  assert {forall k:int. 0 <= k < n ->
          let ck = column_start_index k in
          let rk = row_start_index k in
          let cn = column_start_index n in
          let rn = row_start_index n in
              (ck <> cn -> rule_1_for_chunk g[n<-Empty] ck 8) /\
              (rk <> rn -> rule_1_for_chunk g[n<-Empty] rk 1) /\
              (ck <> cn -> rule_2_for_chunk g[n<-Empty] ck 8) /\
              (rk <> rn -> rule_2_for_chunk g[n<-Empty] rk 1)
               };
```

Les autres assertions sont les faits qui sont composant de la post-condition voulue, par exemple:

```
assert {forall k:int. 0 <= k < n ->
          let ck = column_start_index k in
          let rk = row_start_index k in
          let cn = column_start_index n in
          let rn = row_start_index n in
              (ck <> cn -> rule_1_for_chunk g ck 8) /\
              (rk <> rn -> rule_1_for_chunk g rk 1) /\
              (ck <> cn -> rule_2_for_chunk g ck 8) /\
              (rk <> rn -> rule_2_for_chunk g rk 1)
               };
```

## 1.7 Proving the Main Algorithm

**Question 21** Pour prouver le programme $solve - aux$, j'ai ajouté la post-condition " $g = (oldg)[n \leftarrow Empty]$ " dans le programme "check-cell-change". J'ai aussi ajouté le lemme

```
subst_inchange : forall g. forall n.
                 g[n] = Empty -> g[n <- Empty] = g
```

ce lemme qui est évidement correct, puisque le contenu de chaque case ne change pas.

Après avoir ajouté la post-condition " $Invalid \rightarrow (g = old)[n \leftarrow Empty]$ dans la fonction $check - cell - change$ grâce à une remarque de M. Claude Marché, toutes les conditions de vérifications du programme $solve - aux$ sont finalement prouvées.