MPRI 2.36.1 — Proof of Programs

Solving Takuzu puzzles

Jules Saget

February 25, 2021

Contents

1 Appetizers: Basic Functions on Arrays **1**
1.1 Check for consecutive zeros 1

A Solving Takuzu Puzzles **2**
A.1 Appetizers 2
A.1.1 Checking if in an array there is never 3 consecutive zeros 2
A.1.2 Checking if an array contains as many zeros and ones . . 4
A.1.3 Checking for identical sub-arrays 4
A.2 Takuzu 5
A.2.1 Takuzu puzzle description 5
A.2.2 Takuzu rules 6
A.2.3 Rules satisfaction for a given cell 8
A.2.4 The main algorithm 10
A.3 Some Tests 11

1 Appetizers: Basic Functions on Arrays

1.1 Check for consecutive zeros

1. Because this implementation is simple and follows the definition closely, choosing the right loop invariant was easy.

A Solving Takuzu Puzzles

MPRI course 2-36-1 Proof of Programs - Project 2020-2021

A.1 Appetizers

Some simple functions on arrays of integers

```
module Appetizers

predicate __FORMULA_TO_BE_COMPLETED__
constant __TERM_TO_BE_COMPLETED__ : 'a
constant __VARIANT_TO_BE_COMPLETED__ : int
let constant __EXPRESSION_TO_BE_COMPLETED__ : int = 0
let constant __CODE_TO_BE_COMPLETED__ : unit = ()

use int.Int
use array.Array
```

A.1.1 Checking if in an array there is never 3 consecutive zeros

QUESTION 1 Specification of the first check

```
predicate no_3_consecutive_zeros_sub (a:array int) (l:int) =
  forall i. 0 <= i < l-2 -> not (a[i] = a[i+1] = a[i+2] = 0)
```

[no_3_consecutive_zeros_sub a l] is true whenever in the sub-array [a[0..l-1]], there are no 3 consecutives occurrences of [0]

```
predicate no_3_consecutive_zeros (a:array int) =
  no_3_consecutive_zeros_sub a (Array.length a)
```

QUESTION 2 implementation 1

```
exception TripleFound

let no_3_consecutive_zeros_version_1 (a : array int) : bool
  ensures { result = True <-> no_3_consecutive_zeros a }
  =
  try
    for i=0 to Array.length a - 3 do
      invariant { no_3_consecutive_zeros_sub a (i+2) }
      if a[i] = 0 && a[i+1] = 0 && a[i+2] = 0 then raise TripleFound;
    done;
    True
  with TripleFound -> False
  end
```

QUESTION 3 implementation 2

```
let no_3_consecutive_zeros_version_2 (a : array int) : bool
  ensures { result = True <-> no_3_consecutive_zeros a }
  =
  if a.length < 3 then True else
  let ref last2 = a[0] in
  let ref last1 = a[1] in
  try
    for i=2 to Array.length a - 1 do
      invariant {
        last1=a[i-1] /\
        last2=a[i-2] /\
        no_3_consecutive_zeros_sub a i
      }
      let v = a[i] in
      if v = 0 && last1 = 0 && last2 = 0 then raise TripleFound;
      last2 <- last1;
      last1 <- v;
    done;
    True
  with TripleFound -> False
  end
```

QUESTION 4 implementation 3

```
let no_3_consecutive_zeros_version_3 (a : array int) : bool
  ensures { result = True <-> no_3_consecutive_zeros a }
  =
  let ref count_zeros = 0 in
  try
    for i=0 to Array.length a - 1 do
      invariant {
        0 <= count_zeros <= 2 /\
        count_zeros <= i /\
        (count_zeros = 1 -> a[i-1] = 0) /\
        (i > 0 -> (a[i-1] = 0 -> count_zeros >= 1)) /\
        (i > 1 -> (a[i-1] = 0 = a[i-2] <-> count_zeros = 2)) /\
        (a[i] = 0 /\ count_zeros = 2 -> not no_3_consecutive_zeros_sub a (i+1)) /\
        no_3_consecutive_zeros_sub a i
        }
      if a[i] = 0 then
        if count_zeros = 2 then raise TripleFound
        else count_zeros <- count_zeros + 1
      else count_zeros <- 0
    done;
    True
  with TripleFound -> False
  end
```

A.1.2 Checking if an array contains as many zeros and ones

QUESTION 5

```
let rec ghost 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`

```
  variant { __VARIANT_TO_BE_COMPLETED__ }
  = if __FORMULA_TO_BE_COMPLETED__ then 0 else
    if __FORMULA_TO_BE_COMPLETED__ then 1 + num_occ e f i (j-1) else num_occ e f i (j-1)
```

QUESTIONS 6 and 7

```
let count_number_of (e:int) (a:array int) : int
  ensures { __FORMULA_TO_BE_COMPLETED__ }
  =
  let ref n = 0 in
  for i=0 to a.length - 1 do
    invariant { __FORMULA_TO_BE_COMPLETED__ }
    __CODE_TO_BE_COMPLETED__
  done;
  n

let same_number_of_zeros_and_ones (a:array int) : bool
  ensures { result = True <-> num_occ 0 a.elts 0 a.length = num_occ 1 a.elts 0 a.length }
  =
  count_number_of 0 a = count_number_of 1 a
```

[same_number_of_zeros_and_ones a] returns [true] when [a] contains exactly the same number of occurrences of [0] and of [1]

A.1.3 Checking for identical sub-arrays

QUESTION 8

```
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

```
= forall k:int. __FORMULA_TO_BE_COMPLETED__
```

QUESTION 9

```
exception DiffFound

let check_identical_sub_arrays (a:array int) (o1 o2 l:int) : bool
  requires { __FORMULA_TO_BE_COMPLETED__ }
  ensures { result = True <-> identical_sub_arrays a o1 o2 l }
= try
    for k=0 to l-1 do
      invariant { __FORMULA_TO_BE_COMPLETED__ }
```

```
      if a[o1+k] <> a[o2+k] then raise DiffFound
    done;
    True
  with DiffFound -> false
  end

end
```

A.2 Takuzu

```
module Takuzu

use int.Int
use array.Array
use int.ComputerDivision

predicate __FORMULA_TO_BE_COMPLETED__
constant __TERM_TO_BE_COMPLETED__ : 'a
constant __VARIANT_TO_BE_COMPLETED__ : int
let constant __EXPRESSION_TO_BE_COMPLETED__ : int = 0
let constant __CODE_TO_BE_COMPLETED__ : unit = ()
```

A.2.1 Takuzu puzzle description

```
type elem = Zero | One | Empty

let eq (x y : elem) : bool ensures { result = True <-> x = y }
= match x,y with
| Empty,Empty
| One,One
| Zero,Zero -> True
| _ -> False
end

type takuzu_grid = array elem

let function column_start_index (n:int) : int = mod n 8
let function row_start_index (n:int) : int = 8*(div n 8)

predicate valid_chunk (s i:int) =
  (i = 1 /\ 0 <= s <= 56 /\ mod s 8 = 0) \/ (i = 8 /\ 0 <= s <= 7)

lemma valid_chunk :
  forall s i. valid_chunk s i ->
    forall k. 0 <= k < 8 -> 0 <= s + k*i < 64

function acc (g:takuzu_grid) (start incr k : int) : elem = g[start+incr*k]

let acc (g:takuzu_grid) (start incr k : int) : elem
  requires { g.length = 64 }
```

```
  requires { valid_chunk start incr }
  requires { 0 <= k < 8 }
  ensures { result = acc g start incr k }
=
  g[start+incr*k]
```

A.2.2 Takuzu rules

```
exception Invalid
```

QUESTION 10 Rule 1 for chunks

```
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

```
   forall k:int. __FORMULA_TO_BE_COMPLETED__
```

```
predicate rule_1_for_chunk (g:takuzu_grid) (start incr:int) =
```

`rule_1_for_chunk g s i` is true when rule 1 is not violated in chunk `(s,i)` of grid `g`

```
  no_3_consecutive_identical_elem g start incr 8
```

QUESTION 11

```
let check_rule_1_for_chunk (g:takuzu_grid) start incr
```

`check_no_3_consecutive_identical_elements g s i` check whether the chunk `(s,i)` in grid `g` is satisfiable

```
  requires { g.length = 64 }
  requires { valid_chunk start incr }
  ensures { rule_1_for_chunk g start incr }
  raises { Invalid -> true }
=
  let ref count_zeros = 0 in
  let ref count_ones = 0 in
  for i=0 to 7 do
      invariant { __FORMULA_TO_BE_COMPLETED__ }
    match acc g start incr i with
      | Zero ->
        if count_zeros = 2 then raise Invalid else
           begin count_zeros <- count_zeros + 1; count_ones <- 0 end
      | One ->
        if count_ones = 2 then raise Invalid else
           begin count_ones <- count_ones + 1; count_zeros <- 0 end
      | Empty -> count_zeros <- 0; count_ones <- 0
    end
  done
```

{QUESTION 12}
Rule 2 for chunks

```
let rec function num_occ (e:elem) (g:takuzu_grid) (start incr:int) (l:int)
```

`num_occ e g start incr l` denotes the number of occurrences of `e` in the `l` first elements of the chunk `(start,incr)` of the grid `g`

```
  requires { __FORMULA_TO_BE_COMPLETED__ }
  variant { __VARIANT_TO_BE_COMPLETED__ }
= (* CODE TO BE COMPLETED *) 0
```

```
let count_number_of (e:elem) (g:takuzu_grid) start incr : int
```

`count_number_of e g start incr` returns the number of occurrences of `e` in the chunk `(start,incr)` of the grid `g`

```
  requires { __FORMULA_TO_BE_COMPLETED__ }
  ensures { result = num_occ e g start incr 8 }
  =
  let ref n = 0 in
  for i=0 to 7 do
    invariant { __FORMULA_TO_BE_COMPLETED__ }
    if eq (acc g start incr i) e then n <- n+1
  done;
  n
```

{QUESTION 13}

```
predicate rule_2_for_chunk (g:takuzu_grid) (start incr:int) =
```

`rule_2_for_chunk g s i` is true when rule 2 is not violated in chunk `(s,i)` of grid `g`

```
  num_occ Zero g start incr 8 <= __TERM_TO_BE_COMPLETED__ /\
  __FORMULA_TO_BE_COMPLETED__
```

```
let check_rule_2_for_chunk (g:takuzu_grid) start incr : unit
  requires { g.length = 64 }
  requires { valid_chunk start incr }
  ensures { rule_2_for_chunk g start incr }
  raises { Invalid -> true }
  =
  if count_number_of Zero g start incr > __EXPRESSION_TO_BE_COMPLETED__ then raise Invalid
  __CODE_TO_BE_COMPLETED__
```

{QUESTION 14}
Rule 3 for chunks

```
predicate identical_chunks (g:takuzu_grid) (s1 s2:int) (incr:int) (l:int)
```

`identical_chunks g s1 s2 i` is true whenever the chunks `(s1,i)` and `(s2,i)`, in their first `l` elements, have no empty cells and are pointwise identical

```
= forall k. 0 <= k < l ->
    __FORMULA_TO_BE_COMPLETED__

exception DiffFound

let check_identical_chunks g start1 start2 incr : bool
  requires { __FORMULA_TO_BE_COMPLETED__ }
  ensures { result = True <-> identical_chunks g start1 start2 incr 8 }
= try
    for i=0 to 7 do
      invariant { __FORMULA_TO_BE_COMPLETED__ }
      match acc g start1 incr i, acc g start2 incr i with
      | Zero,Zero -> __CODE_TO_BE_COMPLETED__
      | One,One -> __CODE_TO_BE_COMPLETED__
      | _ -> __CODE_TO_BE_COMPLETED__
      end
    done;
    True
  with DiffFound -> False
  end
```

{QUESTION 15}

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

let check_rule_3_for_column (g:takuzu_grid) (start:int) : unit
  requires { __FORMULA_TO_BE_COMPLETED__ }
  ensures { forall k. 0 <= k < 8 /\ k <> start ->
              not (identical_columns g start k) }
  raises { Invalid -> true }
=
  for i=0 to 7 do
    invariant { __FORMULA_TO_BE_COMPLETED__ }
      (* CODE TO BE COMPLETED *)raise Invalid
  done

predicate identical_rows (g:takuzu_grid) (s1 s2:int) =
  identical_chunks g s1 s2 1 8

let check_rule_3_for_row (g:takuzu_grid) (start:int) : unit
  requires { __FORMULA_TO_BE_COMPLETED__ }
  ensures { forall k. 0 <= k < 8 /\ 8*k <> start ->
              not (identical_rows g start (8*k)) }
  raises { Invalid -> true }
= (* CODE TO BE COMPLETED *)raise Invalid
```

A.2.3 Rules satisfaction for a given cell

QUESTION 16

```
predicate rule_1_for_cell (g:takuzu_grid) (n:int) =
```

`rule_1_for_cell g n` is true whenever the first Takuzu rule is satisfied for the row and the column of the cell number `n`

```
  let cs = column_start_index n in
  let rs = row_start_index n in
  __FORMULA_TO_BE_COMPLETED__
```

```
predicate rule_2_for_cell (g:takuzu_grid) (n:int) =
```

`rule_2_for_cell g n` is true whenever the second Takuzu rule is satisfied for the row and the column of the cell number `n`

```
  let cs = column_start_index n in
  let rs = row_start_index n in
  __FORMULA_TO_BE_COMPLETED__
```

```
predicate rule_3_for_cell (g:takuzu_grid) (n:int) =
```

`rule_3_for_cell g n` is true whenever the third Takuzu rule is satisfied for the row and the column of the cell number `n`

```
  let cs = column_start_index n in
  let rs = row_start_index n in
  forall i. 0 <= i < 8 -> __FORMULA_TO_BE_COMPLETED__
```

```
predicate valid_for_cell (g:takuzu_grid) (i:int) =
```

`valid_for_cell g n` is true whenever cell number `n` satisfy the Takuzu rules

```
  rule_1_for_cell g i /\ rule_2_for_cell g i /\ rule_3_for_cell g i
```

```
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

```
= forall i. 0 <= i < n -> valid_for_cell g i
```

QUESTION 17

```
let check_at_cell (g:takuzu_grid) (n:int) : unit
```

`check_at_cell g n` returns normally if the grid `g` satisfy the rules for cell `n`.

```
  requires { __FORMULA_TO_BE_COMPLETED__ }
  ensures { valid_for_cell g n }
  raises { Invalid -> true }
=
  let col_start = column_start_index n in
```

```
  let row_start = row_start_index n in
  check_rule_1_for_chunk g col_start 8;
  check_rule_1_for_chunk g row_start 1;
  check_rule_2_for_chunk g col_start 8;
  check_rule_2_for_chunk g row_start 1;
  check_rule_3_for_column g col_start;
  check_rule_3_for_row g row_start
```

QUESTIONS 18, 19 AND 20

```
let check_cell_change (g:takuzu_grid) (n:int) (e:elem) : unit
```

check_cell_change g n e takes a grid g that satisfies the rules up to cell n (not included). it sets cell n to the given value e and checks if the rules are still satisfied for cell n and returns normally. It raises exception Invalid if any check fails. It should be used incrementally, as it assumes that the rules are already satisfied for cell whose number is strictly smaller than n.

```
  requires { __FORMULA_TO_BE_COMPLETED__ }
  requires { valid_up_to (g[n<-Empty]) n }
  writes { g }
  ensures { valid_up_to g (n+1) }
  raises { Invalid -> true }
=
  g[n] <- e;
  assert { valid_up_to g[n<-Empty] n };
  check_at_cell g n
```

A.2.4 The main algorithm

```
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

```
= forall k. 0 <= k < n -> g[k] <> 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.

```
= forall k. 0 <= k < 64 -> g1[k] <> Empty -> g2[k] = g1[k]
```

QUESTION 21

```
exception SolutionFound

let rec solve_aux (g:takuzu_grid) (n:int) : unit
  requires { __FORMULA_TO_BE_COMPLETED__ }
  requires { full_up_to g n }
  requires { valid_up_to g n }
  writes { g }
```

```
  variant { __VARIANT_TO_BE_COMPLETED__ }
  ensures { __FORMULA_TO_BE_COMPLETED__ }
  raises { SolutionFound -> extends (old g) g /\ full_up_to g 64 /\ valid_up_to g 64 }
=
  if n=64 then raise SolutionFound;
  match g[n] with
  | Zero | One ->
    try
      check_at_cell g n; solve_aux g (n+1)
    with Invalid -> ()
    end
  | Empty ->
    try
      check_cell_change g n Zero;
      solve_aux g (n+1)
    with Invalid -> ()
    end;
    try
      check_cell_change g n One;
      solve_aux g (n+1)
    with Invalid -> ()
    end;
    g[n] <- Empty
  end

exception NoSolution

let solve (g:takuzu_grid) : unit
  requires { g.length = 64 }
  ensures { full_up_to g 64 }
  ensures { extends (old g) g }
  ensures { valid_up_to g 64 }
  raises { NoSolution -> true }
=
  try
    solve_aux g 0;
    raise NoSolution
  with SolutionFound -> ()
  end

end
```

A.3 Some Tests

```
module Test

  use array.Array
  use Takuzu

  let empty () : takuzu_grid
```

```
  raises { NoSolution -> true }
= let a = Array.make 64 Empty in
  Takuzu.solve a;
  a
```

Solving the empty grid: easy, yet not trivial
Other examples

```
let example1 ()
  raises { NoSolution -> true }
= let a = Array.make 64 Empty in
  a[2] <- Zero;
  a[5] <- One;
  a[8] <- One;
  a[22] <- Zero;
  a[25] <- Zero;
  a[27] <- Zero;
  a[28] <- Zero;
  a[30] <- Zero;
  a[41] <- Zero;
  a[42] <- Zero;
  a[44] <- Zero;
  a[50] <- Zero;
  a[52] <- One;
  a[56] <- One;
  a[62] <- Zero;
  a[63] <- Zero;
  Takuzu.solve a;
  a

let example2 ()
  raises { NoSolution -> true }
= let a = Array.make 64 Empty in
  a[4] <- Zero;
  a[8] <- One;
  a[13] <- Zero;
  a[14] <- One;
  a[22] <- One;
  a[25] <- One;
  a[28] <- One;
  a[33] <- One;
  a[46] <- Zero;
  a[47] <- Zero;
  a[52] <- One;
  a[55] <- Zero;
  a[57] <- Zero;
  a[58] <- Zero;
  a[60] <- One;
  Takuzu.solve a;
  a
```

```
let example3 ()
    raises { NoSolution -> true }
  = let a = Array.make 64 Empty in
    a[1] <- Zero;
    a[3] <- Zero;
    a[7] <- Zero;
    a[12] <- One;
    a[18] <- One;
    a[23] <- Zero;
    a[25] <- One;
    a[37] <- One;
    a[40] <- Zero;
    a[46] <- Zero;
    a[51] <- One;
    a[53] <- Zero;
    a[54] <- Zero;
    a[57] <- Zero;
    a[60] <- One;
    Takuzu.solve a;
    a

let example4 ()
    raises { NoSolution -> true }
  = let a = Array.make 64 Empty in
    a[1] <- One;
    a[2] <- One;
    a[5] <- One;
    a[7] <- Zero;
    a[9] <- Zero;
    a[11] <- Zero;
    a[21] <- One;
    a[23] <- Zero;
    a[34] <- Zero;
    a[38] <- One;
    a[40] <- Zero;
    a[44] <- Zero;
    a[47] <- Zero;
    a[53] <- One;
    a[55] <- One;
    a[56] <- Zero;
    Takuzu.solve a;
    a

let example5 ()
    raises { NoSolution -> true }
  = let a = Array.make 64 Empty in
    a[7] <- Zero;
    a[15] <- One;
    a[21] <- Zero;
```

```
    a[24] <- Zero;
    a[39] <- Zero;
    a[45] <- One;
    a[46] <- One;
    a[50] <- One;
    a[54] <- One;
    a[56] <- One;
    a[59] <- Zero;
    a[60] <- Zero;
    Takuzu.solve a;
    a

let example6 ()
    raises { NoSolution -> true }
  = let a = Array.make 64 Empty in
    a[0] <- One;
    a[2] <- One;
    a[7] <- One;
    a[11] <- One;
    a[20] <- Zero;
    a[30] <- One;
    a[32] <- One;
    a[37] <- Zero;
    a[47] <- Zero;
    a[50] <- One;
    a[53] <- Zero;
    a[54] <- One;
    a[57] <- Zero;
    a[58] <- Zero;
    a[62] <- One;
    Takuzu.solve a;
    a

end
```

Generated by why3doc 1.3.3