EXTENDING LEAN CELLULAR AUTOMATA FRAMEWORK – BOUNDARY CONDITIONS AND PROPERTIES OF CANONICAL FORMS

František SILVÁŠI, Martin TOMÁŠEK
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice,
Letná 9, 042 00 Košice, E-mail: frantisek.silvasi@tuke.sk, martin.tomasek@tuke.sk

ABSTRACT
We present several extensions to our Lean–based formal mechanized framework for computing with cellular automata (CA). First we extend the definition of CAs to allow for specification of arbitrary boundary conditions. We use this addition to represent constant and periodic boundaries. We then formulate and prove numerous missing properties pertaining to canonical forms of cellular automata, culminating in a theorem stating that canonical forms preserve counts of nonempty cell states.

Keywords: *Cellular Automata, Formalization, Lean, Mechanization*

1. INTRODUCTION

Our original work [1] represents the first step towards building a formalized environment for reasoning about cellular automata. This study is a direct continuation of the effort, focused both on extending it in ways that are common in non–mechanical theories of this kind, e.g. allowing representation of boundary conditions, as well as formulating and proving various important properties about canonical forms that we utilise to simplify the way users interact with the system.

The paper is structured as follows. The first section provides context of the study as well as articulates our goals in a more explicit manner. Section 2 explains the way how the framework is extended to allow for specification of custom boundary conditions. In the third section, we first outline basic definitions from the original work about which we then formulate and prove various theorems. The last section is a summary of the study as well as an outline of possible future work.

We shall now provide a short overview of the state of the art of the area. However, we would like to kindly direct the interested reader to the introductory part of our original paper for more complete context.

To the best of our knowledge, no other CA mechanization exists. It is therefore difficult to evaluate our approach in the context of existing solutions. Nevertheless, we find it important to mention that automata in general have been investigated in mechanical frameworks. For example, finite state automata have been formalized in Coq by Fillitre [2] and their connection with regular languages has been further explored mechanically by Doczkal [3].

It is pertinent to briefly outline how our formalization differs from its abstract counterpart [4]. We shall digress for just a moment to once again refer the reader to our original study, namely section 5, for further details on this topic. Our original work represents a formalization with great emphasis on preservation of computability – this allows us to use the Lean code both as a CA simulator as well as a mechanical environment to reason about their properties. These uses are symbiotic in that we can use certified computation to help us with proofs and in turn, we can use proven–correct properties to optimize computation.

These benefits do come at a price however. We have lost some faithfulness to the original model. This manifests the most in the presence of the so called "extension function" which arbitrarily extends underlying lattices so that computation is not limited to a bounded space. Its presence interferes with properties of automata that follow naturally from the abstract counterpart – as such, we have introduced canonical forms that alleviate some of these issues.

This paper formulates several important properties about said extension functions and canonical forms, ultimately building up to a mechanized formal proof of the statement $\forall a\ x, x \neq a.empty \rightarrow count\ a\ x = count\ (make\_canonical\ a)\ x$. Various important properties need to be shown along the way, as shall be demonstrated shortly.

Additionally, we also extend the framework in a way such that it is possible (and convenient) to define custom boundary conditions, therefore replacing the original behaviour which enforced infinite lattices with empty constant quasi–boundaries.

The entire formalization is available at `https://github.com/frankSil/CAExtensions`. For ease of reuse, we use the Lean packaging manager – once one installs Lean 3.4.2, it is only necessary to run *leanpkg build* from the root of the repository. The formalization is complete in the sense that all proofs have been checked by Lean.

2. CUSTOM BOUNDARY CONDITIONS

For brevity, we shall only include the altered Lean code and focus mostly on discussing the introduced differences. However, for context, we will give a brief outline of the functionality in general.

```
structure cautomaton (α : Type)
                     [decidable_eq α] :=
  (g     : vec_grid₀ α)
  (empty : α)
  (neigh : point → list point)
  (bound : (bounding_box → bounding_box) ⊕
           (α → vec_grid₀ α → point → α))
  (f     : α → list α → α)
```

The first component of a cellular automaton is a two dimensional lattice *g*. The field *empty* represents the empty cell state, *neigh* is a neighbourhood function and *f* is the automaton rule acting on local configurations. Originally, the member *bound* was called *ext* and represented an extension function that would shrink or expand the grid based the needs of the function *f*. Now we have added the possibility to either provide an extension function, or instead give a rule for boundary condition. If one opts into the latter, it is implicitly assumed that no extension of the underlying lattice happens. The boundary itself is a function $(notin : \alpha) \to (lattice : vec\_grid_0\ \alpha) \to (p : point) \to \alpha$. The idea behind it is that we allow for two kinds of behaviours, depending on whether $p \in lattice$. The first argument *notin* allows us to specify a default value, generally used when $p \notin lattice$.

We also define two commonly used boundary situations.

```
def bound_const {α : Type*}
  [grid α] (empty : carrier α)
  (g : α) (p : point) : carrier α :=
  if h : p ∈ g
  then abs_data g p
  else empty
```

This is the most straightforward boundary situation, in which every position outside of underlying lattice is defined to be some constant *empty*. The function *abs_data g p* simply returns a cell of lattice *g* on position *p*. Also, we use *gbl* and *gtr* to respectively represent bottom left and top right corners of grids – note that *gtr* can be easily computed from *gbl* and *rows g/cols g* . The rest of notation should be self–explanatory but as with everything throughout this study, one can consult the original work for further information.

```
def bound_periodic {α : Type*}
  [grid α] (empty : carrier α)
  (g : α) (p : point) : carrier α :=
  if h : p ∈ g
  then abs_data g p
  else
  if b₁ : (gbl g).y ≤ p.y ∧ p.y < (gtr g).y
  then
    if p₁ : p.x ≥ (gtr g).x
    then abs_data g P₁
    else abs_data g P₂
  else
    if b₂ : (gbl g).x ≤ p.x ∧ p.x < (gtr g).x
    then
      if p₂ : p.y ≥ (gtr g).y
      then abs_data g P₃
      else abs_data g P₄
    else empty
```

Where

$P_1 = \langle \mathrm{p.y}, (\mathrm{gbl\ g}).\mathrm{x} + (\mathrm{p.x} - (\mathrm{gbl\ g}).\mathrm{x})\ \%\ (\mathrm{cols\ g})\rangle$
$P_2 = \langle \mathrm{p.y}, (\mathrm{gtr\ g}).\mathrm{x} - 1 - ((\mathrm{gbl\ g}).\mathrm{x} - \mathrm{p.x} - 1)\ \%\ (\mathrm{cols\ g})\rangle$
$P_3 = \langle (\mathrm{gbl\ g}).\mathrm{y} + (\mathrm{p.y} - (\mathrm{gbl\ g}).\mathrm{y})\ \%\ (\mathrm{rows\ g}),\ \mathrm{p.x}\rangle$
$P_4 = \langle (\mathrm{gtr\ g}).\mathrm{y} - 1 - ((\mathrm{gbl\ g}).\mathrm{y} - \mathrm{p.y} - 1)\ \%\ (\mathrm{rows\ g}),\ \mathrm{p.x}\rangle$

The function *abs_data g p* requires a proof of $p \in g$, which is just notation for $(gbl\ g).x < p.x \leq (gtr\ g).x \wedge (gbl\ g).y \leq p.y < (gtr\ g).y$. For brevity, we just note that it is relatively straightforward to show the necessary properties for all P_n. Said proofs are of course formalized in full and we invite the interested reader to inspect the enclosed formalization.

This boundary condition specifies periodic boundaries in the sense that "leaving" the lattice from one side has the effect of entering the lattice on the opposite side. This corresponds spatially with a torus.

Having modified the definition of cellular automaton itself, we also need to slightly alter the way we compute with them. The function *ext_aut a* that expands underlying lattices is changed in a way such that it acts as the identity function in case a boundary condition has been specified. Otherwise it uses the supplied extension function, i.e. the left part of *a.bound*. On the other hand, the function *next_gen a* that computes future generations of configurations checks if an extension function has been supplied. If so, it uses *bound_const a.empty*. Otherwise we just utilize the provided boundary function, i.e. the second component of the sum *a.bound*.

These changes are fairly trivial and therefore do not warrant an inclusion in–full within the paper. Please do consult the enclosed git repository. As a tiny side note, we have modified Lengton's Ant [5] to use periodic boundaries to demonstrate the way it behaves – the file *ant.lean* contains the definition along with a preset configuration called *simple*. For various values of *k*, calling `#eval step_n simple k` allows one to observe the behaviour of this boundary.

3. PROPERTIES OF CANONICAL FORMS

3.1. Exposition and context

Colloquially, we say an automaton is in canonical form if it contains no empty rows nor columns on the edges of its lattice. Formally, we have a canonicalization procedure $make\_canonical : cautomaton\ \alpha \to cautomaton\ \alpha$ and define that an automaton *a* is in a canonical form if and only if $make\_canonical\ a = a$. Also, configurations of automata are stored in grids, which are important for the study only conceptually – one can think of them as geometrical orthogonal two–dimensional lattices implemented as binary functions that associate a unique cell state to each row and column pair. Let us copy verbatim the most important definitions we shall be using throughout the study.

Definition 3.1. *A subgrid is a part of a grid g delineated by a bounding box bb within the bounds of g.*

```
def subgrid {α : Type*}
  [grid α]
  (g : α) (bb : bounding_box)
  (h : overlaid_by bb g) :
  fgrid₀ (carrier α) :=
⟨rows_of_box bb, cols_of_box bb, ..., bb.p₁,
 λx y, abs_data g ⟨⟨x.1, ...⟩, ⟨y.1, ...⟩⟩⟩
```

Note that the definition is trivial. It simply delegates to the underlying original grid while restricting elements that are outside of the provided bounding box. We shall also be using *overlaid_by a b* to mean that *a* fits spatially into *b* and we also have a function *gip_g* that is defined to be the exhaustive sequence of row–major coordinates starting at the bottom left within *g*.

Also note that we can use *gip_g* and *abs_data* to define a function that turns any grid into a list of its elements. We shall henceforth denote such function with a unary prefix symbol $\wp$ and its definition is as follows.

```
def ℘ {α : Type*} [grid α] (g : α) :=
  map (abs_data g ∘ grid_point_of_prod
                  ∘ inject_into_bounded g)
      (attach $ gip g)
```

The functions *attach*, *grid_point_of_prod* and *inject_into_bounded* are mostly superfluous and do some type–fixing for Lean; they can be safely ignored and we shall be omitting them throughout the study.

Definition 3.2. *Compute_bounds represents the smallest possible bounding box containing all nonempty elements of a lattice.*

```
def compute_bounds {α : Type}
  [decidable_eq α] (a : cautomaton α) :
  bounding_box :=
  let bounded  := gip_g a.g in
  let mapped   := ℘ a.g in
  let zipped   := zip bounded mapped in
  let filtered :=
    filter (λx, snd x ≠ a.empty) zipped in
  if h : empty_list filtered
  then ⟨
    gbl a.g, gtr a.g, grid_is_bounding_box
  ⟩
  else
  let unzipped := fst ∘ unzip $ filtered in
  let xs       := map point.x unzipped in
  let ys       := map point.y unzipped in
  let min_x    := min_element xs ... in
  let max_x    := max_element xs ... in
  let min_y    := min_element ys ... in
  let max_y    := max_element ys ... in
  ⟨⟨min_x, min_y⟩, ⟨max_x + 1, max_y + 1⟩,
  ...⟩
```

The function simply finds extrema for non–empty cells and computes the smallest bounding box containing them all. For space–preserving reasons, we shall denote $compute\_bounds = cb$.

Definition 3.3. *The function make_canonical a represents a new automaton with all properties equivalent to a but with its underlying lattice restricted in accordance with cb.*

```
def make_canonical : cautomaton α :=
  {a with g :=
   (subgrid a.g (cb a) P)}
```

The only property in our original work on *cb* is that the bounding box it produces fits within the bounds of the original grid (note the *P* in the preceding definition). It turns out it is insufficient. The main motivation behind the general counting lemma we shall now prove comes from an attempt to show that a particular lattice–gas automaton hpp [7] preserves the number of gas molecules as time progresses[1].

3.2. Formulation of properties with their respective proofs

We formulate and prove the main result first. On the way to prove it, we discover and prove various useful properties about grids, their subgrids and canonical forms of automata. We designate general–purpose properties as theorems and always note what they are used for. Everything else is marked as a lemma and can be considered to be of technical nature. We shall be omitting several simpler facts and we do encourage the interested reader to inspect the enclosed github repository in case anything is unclear. At the end of the paper, we enclose formulations of lemmas used from our baseline study.

Henceforth, unless specified otherwise, let α be any Type with decidable equality defined thereover and let $a : cautomaton\ \alpha$. Also, we will occasionally use grids, bounding boxes and their corresponding *gbl* / *gtr* corners interchangeably.

Theorem 3.1. *Given a nonempty $x : \alpha$, count of x in a is preserved by make_canonical.*

Proof. Counting x in an automaton a is by definition equivalent to counting x in $\wp\, a.g$. Considering also the definition 3.3, it suffices to show

$$count\ x\ (\wp\, a.g) = count\ x\ \wp\,(subgrid\ a.g\ (cb\ a)\ ...)$$

which follows directly from lemma 3.1. □

Definition 3.4. *Effectively the dual of subgrid. Given a grid g and a bounding box bb within g, return all elements of g that are not in bb.*

```
def subgrid' {α : Type*} [grid α]
  (bb : bounding_box) (h : overlaid_by bb g)
  : list (carrier α) :=
  map (abs_data g ∘ inject_filter_bounded _)
    (attach $ filter (λp, p ∉ bb)
    (gip_g g))
```

As with $\wp$, note here that *inject_filter_bounded* and *attach* once again just help Lean accept our definitions. We shall for the most part ignore their presence. Also, we will not write the last explicit argument for this definition, which also applies for *subgrid*.

Lemma 3.1. *For any nonempty $x : \alpha$, $count\ x\ (\wp\, a.g) = count\ x\ \wp\,(subgrid\ a.g\ (cb\ a))$.*

Proof. Let $bb = cb\ a$ and note that bb is overlaid by $a.g$. It then suffices to show

$$\begin{aligned} &count\ x\ \wp\,(subgrid\ a.g\ (cb\ a)) + \\ &count\ x\ (subgrid'\ a.g\ (cb\ a)) = \\ &count\ x\ \wp\,(subgrid\ a.g\ (cb\ a)) \end{aligned}$$

by theorem 3.3. However, the second summand here is zero by theorem 3.2. □

Theorem 3.2. *For any nonempty $x : \alpha$, $count\ x\ (subgrid'\ a.g\ (cb\ a)) = 0$.*

This result states that *cb* never removes a nonempty cell.

[1]A formalization of *hpp* is available in the enclosed github repository and is a part of the original work.

Proof. Counting elements x in a list l is equivalent to first filtering l for those elements that are equal to x and then taking its length. Then upon unfolding the definition 3.4, it is clear it only suffices to show

$$|filter\ (=x)\ (map\ (abs\_data\ a.g)\ (filter\ (\lambda p, p \notin cb\ a)\ (gip\_g\ a.g)))| = 0$$

Furthermore, lists have length zero iff they are empty and a filtered list l by P is empty iff for all elements $x \in l$, $\neg P\ x$. Let us therefore assume we have an arbitrary $x \in l$ here and we have to show $\neg P\ x$. Considering $l = map\ f\ l'$ in this case, it must be the case that there exists $p \in l'$ for which $f\ p = x$. As such, we have $H : p \in filter\ (\lambda p,\ p \notin cb\ a)\ (gip\_g\ a.g)$ and we need to show $x \neq abs\_data\ a.g\ p$. Assume however for the sake of contradiction that it is not so. First show $abs\_data\ a.g\ p = a.empty$. This is because from H it is clear that both $p \in gip\_g\ a.g$ and $p \notin cb\ a$ and as shown in our previous work (theorem 7), $p \in gip\_g\ a.g$ also entails $p \in a.g$. The property now follows from lemma 3.2. Here we can easily derive a contradiction as recall that the original assumption states $x \neq empty$. ☐

Lemma 3.2. *Given a point $p \in a.g$ and assuming $p \notin cb\ a$, $abs\_data\ a.g\ p = a.empty$*

Proof. First define

$$i = (p.y - (gbl\ a.g).y) * cols\ a.g + (p.x - (gbl\ a.g).x)$$

It follows from $p \in g$ that $0 \leq i < |a.g|$. Then define

$$l = filter\ (\lambda x, x.snd \neq a.empty)\ (zip\ (gip\_g\ a.g)\ (\wp\ a.g))$$

Now assume that l is empty, from which we have that for any p and b such that $(p,\ b) \in zip\ (gip\_g\ a.g)\ (\wp\ a.g)$, $b = a.empty$. Therefore, it suffices to show

$$(p,\ abs\_data\ a.g\ p) \in zip\ (gip\_g\ a.g)\ (\wp\ a.g)$$

and further

$$(p,\ abs\_data\ a.g\ p) = n^{th}\ (zip\ (gip\_g\ a.g)\ (\wp\ a.g))\ i$$

Note this is valid because the size of the zipped lists is equivalent to either of them, which are in turn equivalent to $|a.g|$, which is strongly preceded by i. It now only suffices to exhibit each n^{th} part of the *zip*ped list separately. For

$$n^{th}\ (gip\_g\ a.g)\ i = p$$

we get

$$((gbl\ a.g).x + i\%(cols\ a.g),\ (gbl\ a.g).y + i/(cols\ a.g)) = p$$

by theorem 8 of the preceding study. We are finished with a bit of modular arithmetic. For the second list, we need to show

$$n^{th}\ (\wp\ a.g)\ i = abs\_data\ a.g\ (x,\ y)$$

By lemma 10 of the base study, it suffices to show

$$\begin{aligned}&abs\_data\ a.g\ (\\&\quad (gbl\ a.g).y + i/cols\ a.g,\\&\quad (gbl\ a.g).x + i\%cols\ a.g) = abs\_data\ a.g\ (y,\ x)\end{aligned}$$

which is straightforward to do after the arguments are isolated. In the case where l is not empty, first define

$$l' = (unzip\ (filter\ (\lambda x, x.snd \neq a.empty)\ (zip\ (gip\_g\ a.g)\ (\wp\ a.g)))).fst$$

Note that the assumption $p \notin cb\ a$ is possible in four cases given a nonempty l.

$$p.y < min\_element\ (map\ point.y\ l') \quad (1)$$
$$1 + max\_element\ (map\ point.y\ l') \leq p.y \quad (2)$$
$$p.x < min\_element\ (map\ point.x\ l') \quad (3)$$
$$1 + max\_element\ (map\ point.x\ l') \leq p.x \quad (4)$$

It is now clear that all of these cases imply that either component of p is not in their respective $map\ point.component\ l'$ list, from which we can show in a similar fashion in all four cases, that $abs\_data\ a.g\ p = a.empty$ as follows. Let us arbitrarily pick (1). As mentioned, it follows therefrom that $y \notin map\ point.y\ l'$ from which it is clear that $p \notin l'$. Let us assume $abs\_data\ a.g\ p \neq a.empty$ and derive a contradiction. Define $elem = abs\_data\ a.g\ p$. Show

$$(p,\ elem) \in filter\ (\lambda x, x.snd \neq a.empty)\ (zip\ (gip\_g\ a.g)\ (\wp\ a.g))$$

which is in the filter iff

$$(p,\ elem) \in zip\ (gip\_a.g)\ (\wp\ a.g) \quad (1)$$
$$elem \neq a.empty \quad (2)$$

of which (2) is obvious and (1) follows from theorem 8 and lemma 10 of the original paper. However, we can now trivially show $p \in l'$ which is a contradiction. The other three cases are similar, which concludes the proof. ☐

Theorem 3.3. *Given a type α which is a grid with elements in carrier α, $g : \alpha$, $bb : bounding\_box$ such that bb is overlaid by g and assuming we have decidable equality over elements, then for all $elem : carrier\ \alpha$, $count\ elem\ (\wp\ g) = count\ elem\ (\wp\ (subgrid\ g\ bb)) + count\ elem\ (subgrid'\ g\ bb)$.*

This theorem simply states that we can always split a grid to two parts with *subgrid* and still preserve the count of elements within. The residuum is then *subgrid'*.

Proof. First express each *count* as an appropriate slice of filtered grid indices. Define $P = \lambda p,\ elem = abs\_data\ g\ p$ and let $indices = gip\_g$. For the left hand side, it follows from the definition of $\wp$ that

$$count\ elem\ (\wp\ g) = |filter\ P\ (gip\_g\ g)|$$

from lemma 3.3 we get

$$count\ elem\ (\wp\ (subgrid\ g\ bb)) = |filter\ P\ (filter\ (\lambda p, p \in bb)\ (gip\_g\ g)))|$$

and the definition 3.4 gives us

$$count\ elem\ (\wp\ (subgrid\ g\ bb)) = \\ |filter\ P\ (filter\ (\lambda p, p \notin bb)\ (gip\_g\ g)))|$$

This is straightforward to prove by induction on $gip\_g\ g$. Note that the filters on the right hand side are mutually exclusive and cover the entire list. $\square$

Before we continue, it is worth stopping for just a moment to outline and discuss some differences, in this proof in particular, between Lean and paper proofs. Recall we ignored some type–mending transformations in the definition 3.4, which are necessary as $abs\_data\ g : \{p : point//p \in g\} \to \beta$ for some β that is the type of elements of g – note the type of said expression is not $point \to \beta$. This has profound implications on the way the proof can be constructed. The function $gip\_g\ g$ yields a list of *point*s. It first has to be lifted to a list of *point*s that are within said list – this very suddenly changes three homogeneous lists (all of which contain points) to three separately typed sequences, corresponding with membership within their respective, possibly filtered lists.

As an example, consider $seq = filter\ (\lambda p, p \notin bb)\ (gip\_g\ g)$ which "normally" has the type *list point*. Upon adjoining it with its natural property of each of the points being members of the sequence (we shall denote the function that does this *attach*), we have $attach\ seq : list\ \{p : point//p \in seq\}$. In this manner, the other two lists are their own types as well.

The theorem then becomes unprovable with direct induction over gip_g – this is because the inductive hypothesis has to be formulated in terms of the filtering predicate $P : \{x//x \in hd :: tl\} \to Prop$[2], making our original predicate unusable as it pertains to the entire list. The idea is to generalize over all predicates and structures, prove by induction the general version and specialize for the concrete solution. The abstracted formulation is too wordy to include in the paper, however we do encourage the interested reader to take a look at "*grid.lean/filter_partition_dependent*".

Lemma 3.3. *For all grid Types α with decidable equality defined over their carrier α, given a grid $g : \alpha$, a bounding box bb overlaid by the grid g and an elem : carrier α, $count\ elem\ (\wp\ (subgrid\ g\ bb)) = |filter\ (\lambda p,\ elem = abs\_data\ g\ p)\ (filter\ (\lambda p, p \in bb)\ (gip\_g\ g)))|$.*

Proof. Define $g' = subgrid\ g\ bb$. Counting *elem* in a sequence is the same as taking the length of the list containing only elements equivalent to *elem*. Then, taking the definition of $\wp$ into consideration, we calculate on the left hand side:

$$|filter\ (\lambda x,\ x = elem)\ (map\ (abs\_data\ g')\ (gip\_g\ g')| = \\ |map\ (abs\_data\ g')\ (filter\ ((\lambda x,\ x = elem) \circ \\ (abs\_data\ g'))\ (gip\_g\ g'))| = \\ |filter\ ((\lambda x,\ x = elem) \circ (abs\_data\ g'))\ (gip\_g\ g')| = \\ |filter\ (\lambda p,\ elem = abs\_data\ g'\ p)\ (gip\_g\ g')|$$

Note that theorem 3.4 gives us $H : filter\ (\lambda p,\ p \in bb)\ (gip\_g\ g) = gip\_g\ g'$. Considering now H and congruence of $filter$, it suffices to show $\forall p\ (a : p \in gip\_g\ g'),\ elem = abs\_data\ g'\ p \iff elem = abs\_data\ g\ p$, which follows from the definition of *subgrid*. $\square$

Theorem 3.4. *For all grid Types α, given a grid $g : \alpha$ and a bounding box bb overlaid by the grid g, $filter\ (\lambda p,\ p \in bb)\ (gip\_g\ g) = gip\_g\ (subgrid\ g\ bb)$.*

This theorem expresses that the elements within a subgrid of a grid g made with a bounding box bb are those that remain after we take only those elements from g which belong to bb. Please note that in the following proof, we refer to the definition of gip_g only conceptually – one can inspect the details in the enclosed formalization or in the text of the paper of the baseline study. Also note that we shall use $range\ a\ b$ to mean the sequence $\{a,\ a+1,\ ...,\ b-1\}$ and $grp\ a\ b\ r$ to mean $\{(a,\ r),\ (a+1,\ r),\ ...,\ (b-1,\ r)\}$.

Proof. Let $p_1 = bb.p_1$ and $p_2 = bb.p_2$. Note as also that $p_1 \nearrow p_2$. It follows from the definition of gip_g that we need to show

$$filter\ (\lambda p, p \in g) \\ (join\ (map\ (grp\ (gbl\ g).x\ (gtr\ g).x) \\ (range\ (gbl\ g).y\ (gtr\ g).y))) = \\ join\ (map\ (grp\ p_1.x\ p_2.x)\ (range\ p_1.y\ p_2.y))$$

Let us now set $l = range\ \ p_1.y\ \ p_2.y$ and $l_1 = range\ (gbl\ g).y\ (gtr\ g).y$. Because bb fits within g and $p_1 \nearrow p_2$, we can write l_1 as concatenation of the following three lists, in the respective order.

$$l_2 = range\ (gbl\ g).y\ p_1.y \\ l_3 = l \\ l_4 = range\ p_2.y\ (gtr\ g).y$$

Here we can represent the left hand side as concatenation of $filter\ (\lambda p, p \in g)\ (join\ (map\ (grp\ (gbl\ g).x\ (gtr\ g).x)\ l_n))$ where $n = \{2,\ 3,\ 4\}$ in left to right order (note that, of course, concatenation is not commutative). Here note the first list is empty, because it would be absurd if it were not so – clearly all of its elements must both be within the bounding box vertically yet the very last index is $p_1.y - 1$. Similar reasoning holds for the third list, which is empty as well. We are therefore left to show that the preceding expression with $n = 3$ is equivalent with $join\ (map\ (grp\ p_1.x\ p_2.x)\ (range\ p_1.y\ p_2.y))$, which is straightforward to do. $\square$

4. DISCUSSION, CONCLUSIONS AND FUTURE WORK

This study is a direct continuation of our previous work – originally, we defined canonical forms to deal with computational aspects of cellular automata proofs. We therefore needed very little in terms of their behaviour and just relied on Lean to run the definitions. As a result, we only proved the absolute necessary properties.

This paper, on the other hand, both extends the work in the sense that we have new functionality – namely, we

[2]Note it is standard to induce over the structure of a list with *hd* being its head and *tl* being the tail.

can now specify boundary conditions, as well as finishes formulating and proving properties missing in the baseline work.

Boundary conditions are an important part of the theory of cellular automata. For example, many traffic simulation models are simplified greatly by introducing periodic boundaries, which removes the need to account for infinite spaces. One can for example examine the Biham–Middleton–Levine model [6]. In addition to extending the framework in a way such that custom boundary conditions can be defined, we have also provided two most common boundary functions – the constant and the periodic boundary, which are the functions *bound_const* and *bound_periodic* respectively.

With regards to missing theorems from the original study, we have formulated and proven several new properties. We have listed the most important ones in the paper, along with outlines of their proofs.

- The theorem 3.4 reformulates the function *subgrid* in terms of the more general function $filter$.
- The theorem 3.3 says that splitting a grid into two disjunct parts preserves the count of any elements within the grid.
- The theorem 3.2 states that the residuum we get after obtaining the canonical form of a grid contains only empty cells.
- The theorem 3.1 states that canonicalization only removes empty cells.

Moreover, we have of course also defined several additional lemmas and auxiliary properties that can be utilized while using our CA framework.

The main results of this study are most important in cases where we want to formulate counting arguments for more specific cellular automata, which leads us to further research. As stated, the motivation was to express preservation of molecules in the lattice–gas cellular automaton *hpp*. This paper constitutes a next step in this direction, because it allows us to completely disregard the effects of *make_canonical*. We are therefore left with proving that the function $translation$, as defined in the file *hpp.lean* does in fact preserve molecule counts. Naturally, one can formulate counting arguments for any other cellular automaton specified within our framework.

This study is a part of a bigger project with several open problems and goals. We encourage the interested reader to further confer the section 6.3 of our original paper to learn more about what venues can be explored with regards to this research.

5. AUXILIARY LEMMAS USED IN PROOFS

Theorem 7 (used in theorem 3.2) states that a point is located within a grid if and only if it is in the list generated by the appropriate *gip_g*.

$$p \in g \iff p \in gip\_g\ g$$

Theorem 8 (used in lemma 3.2) states the intuitively obvious – as mentioned, *gip_g* is the exhaustive list of coordinates in a row–major fashion. As such, the elements follow the row / column pattern as formulated in the theorem.

$$\begin{aligned} & n < |g| \implies \\ & n^{th}\ (gip\_g\ g) = (\\ & \quad (gbl\ g).x + n\ \%\ (cols\ g), \\ & \quad (gbl\ g).y + n/(cols\ g)) \end{aligned}$$

Lemma 10 (used in lemma 3.2) reflects the mirrored nature of *gip_g* and $\wp$ – the n^{th} element of a grid is the element at the position computed by *gip_g*.

$$\begin{aligned} & n < |g| \implies \\ & n^{th}\ (\wp\, g) = abs\_data\ g\ (\\ & \quad (gbl\ g).y + n/(cols\ g), \\ & \quad (gbl\ g).x + n\ \%\ (cols\ g))) \end{aligned}$$

ACKNOWLEDGEMENT

This work was supported by the Slovak Research and Development Agency under the contract No. APVV-15-0055. The paper was supported by project KEGA no. 079TUKE4/2017. This work was supported by FEI TUKE Grant no. FEI-2018-57.

REFERENCES

[1] SILVÁŠI, F. – TOMÁŠEK, M.: Lean Formalization of Bounded Grids and Computable Cellular Automata Defined Thereover. To appear in Science of Computer Programming. 2020.

[2] FILLITRE, J.C.: Finite Automata Theory in Coq - A constructive proof of Kleene's theorem. Ecole Normale and Suprieure Lyon, 1997.

[3] DOCZKAL, C. – KAISER, J. O. – SMOLKA, G.: A Constructive Theory of Regular Languages in Coq. Springer International Publishing. In Certified Programs and Proofs. 2013. 82–97.

[4] KUTRIB, M. – VOLLMAR, R. – WORSCH, Th.: Introduction to the special issue on cellular automata. In Parallel Computing. 23/11. 1997. 1567–1576.

[5] LANGTON, C.G.: Studying artificial life with cellular automata. In Proceedings of the Fifth Annual International Conference. 22/1. 1986. 120–149.

[6] BIHAM, O. – MIDDLETON, A. A. – LEVINE, D.: Self-organization and a dynamical transition in traffic-flow models. In Phys. Rev. A. 46/10. 1992. 124–127.

[7] HARDY, J. – POMEAU, Y. – PAZZIS, D. O.: Time evolution of a twodimensional model system. I. In-

variant states and time correlation functions. In Journal of Mathematical Physics. 14/12. 1973. 1746–1759.

Received January 29, 2020, accepted March 23, 2020

BIOGRAPHIES

Frantisek Silváši is a Ph.D. student at Technical University of Košice working on formal software verification utilizing automated reasoning.

Martin Tomášek received Ph.D. degree in Software and Information Systems in 2005 and currently works as an associate professor at Technical University of Košice. His research interests include concurrency theory, distributed systems, and cloud computing.