ACML

automates cellulaire en caml

API tutorial

how to define your favorite 'bistrouillet'
in a few lines of caml

Guillaume Theyssier — April 1, 2024

Before starting...

This document is about writting OCaml source code to define cellular automata using the ACML API. A reasonable knowledge of both OCaml programming language and cellular automata is assumed.

Each section below describes a valid source file to be opened by the application `acml`. Depending on the way you obtained `acml`, there should be somewhere a directory called '`tutorials`' containing the source file of each section (file name ≈ section title).

It is a good idea to have a look at the API reference each time you discover a new function name in this tutorial.

Contents

1 Hello_world 1

2 Parameters 2

3 Configurations 2

4 Arbitrary_rules 3

5 Randomness 4

1 Hello_world

This example illustrates the very basic way of using the ACML API. It defines the elementary cellular automaton whose local rule is the sum modulo 2 of the cell itself and its left and right neighbours (rule 150).
An ACml source file is nothing else than a valid OCaml source file using the ACML API.

```
open Acml
```

The only thing that ACML will keep from your source at the end is not an Acml object but a function transforming a given Acml object into a new one. This is the key concept. I repeat: the API doesn't want an object but a function transforming an object into the desired one. So let's define a function!

```
let transform a =
```

At this point we don't know anything about what's in the Acml object 'a'. We are going to specify what is important for us and forget about the rest. The way to specify things is mostly imperative.

First we specify dimension, number of states and neighbourhood.

```
  set_dimension a 1;
  set_cardinality a 2;
  set_moore_neighbourhood a 1;
```

Then we set the local rule, which is a function taking an array of int and returning an int. The array contains the value of each neighbour in some order (but the order doesn't matter for this example).

```
  set_rule a (fun x → Array.fold_left (+) 0 x mod 2)
```

Finally, we register our transformation function to the API.

```
let _ = valid transform
```

That's all!

2 Parameters

This example shows how to use parameters. This is one of the benefits of using transformations over objects instead of objects directly.

A parameter is just a function returning a value chosen via the user interface; each time the user wants to test a new parameter value, he (or she) sets the parameter and then asks for a new evaluation of the transformation function.

```
open Acml

let transform a =
```

First we create an int parameter between 2 and 50 (3 by default) which can be controlled via the user interface under the parameter name 'Number of states'

```
  let n = param a "Number of states" 2 50 3 in
    set_cardinality a n;
```

Same idea for dimension and neighbourhood. By the way, we discover another kind of parameter called 'choice_param' and returning a string among a list of possible choices.

```
    set_dimension a (param a "Dimension" 1 2 1);
    let r = param a "Radius" 1 50 1 in
      (match
          choice_param a "neighborhood" [ "Moore"; "von Neumann"]
       with
          "von Neumann" → set_vonneumann_neighbourhood a r
        | _ → set_moore_neighbourhood a r);
```

Now we set the local rule: sum modulo 'n' of the neighbours. The form of the local rule is independant of dimension and neighbourhood but depends on the number of states.

```
    set_rule a (fun x → Array.fold_left (+) 0 x mod n)

let _ = valid transform
```

From the user interface one can now easily explore the algebraic 'sum modulo n' rule for any n (up to 50), any radius (up to 50), and dimension 1 or 2.
There are other parameter types in the API.

3 Configurations

This example shows how to define new types of initial configurations and new ways to modify the current configuration.

```
open Acml

let transform a =
```

We start from the 'addition modulo' rule.

```
let n = param a "Number of states" 2 50 3 in
  set_cardinality a n;
  set_dimension a (param a "Dimension" 1 2 1);
  let r = param a "Radius" 1 50 1 in
    (match
        choice_param a "neighborhood" [ "Moore"; "von Neumann"]
     with
          "von Neumann" → set_vonneumann_neighbourhood a r
       | _ → set_moore_neighbourhood a r);
    set_rule a (fun x → (Array.fold_left (+) 0 x) mod n);
```

Now we add a 'everywhere zero' initial configuration. This configuration type will be selectable through the user interface.

```
    add_init_conf a "Everywhere zero" (Uniform 0);
```

Of course, we can use parameters in initial configurations.

```
    let s = param a "sparsity" 2 50 10 in
      add_init_conf a "Sparse configuration" (General (fun x y → if x mod s = 0 ∧
y mod s = 0 then (n − 1) else 0));
```

Finally, we add what's called a 'modif' (i.e. modification method) in the API: that's a function which is called when a zone is selected with the mouse and which can read and modify the content of selected cells. Such a modif method will be selectable via the user interface. Naturally, parameters can be used in modif methods.

```
      let p = param a "period" 1 50 10 in
        add_modif a "Diagonal periodic" (SimpleZone (fun c x y → ((x +
y) mod p) mod n))

let _ = valid transform
```

4 Arbitrary_rules

This example defines the famous Brian's Brain automaton inspired from the excitable medium metaphor. Two aspects of the API are illustrated here: polymorphic API calls and multiset rules.

```
open Acml
```

To make the definition more readable, we can use any OCaml type for states.

```
type state = Quiescent | Excited | Refractory
```

Then we must declare the set of objects of that type which will form the set of states of the cellular automaton.

```
let alphabet = AlphaArray [|Quiescent; Excited; Refractory|]
```

From that point on, we can use 'polymorphic' versions of functions from the API.

```
let transform a =
  set_dimension a 2;
  set_moore_neighbourhood a 1;
```

We now define the local transition rule of the automaton. Here, states are OCaml values of type 'state' (and not integers). Moreover, the local function defined has two parameters: the first (of type 'state') gives the current state of the cell, and the second of type 'state → int' gives for each state its number of occurrences in the neighborhood.

```
  let transition_rule cell howmany = match cell with
      Excited → Refractory
    | Quiescent when howmany Excited = 2 → Excited
    | _ → Quiescent
  in
```

This local function is registered via the appropriate API call.

```
    poly_set_multiset_rule a ~alphabet transition_rule;
    set_name a "Brian's Brain"

let _ = valid transform
```

5 Randomness

This example shows how to use (pseudo-)randomness in rule definitions and/or configurations.

A cellular automaton can be deterministic or stochastic depending on the definition of the local rule. If you give a deterministic local rule, you get a deterministic CA. If you give a local rule using (pseudo-)randomness, there are two possibilities: either you want to define a stochastic CA, or you want to define a deterministic CA where some transitions are chosen at random.

ACML can deal with all scenarii provided you specify what you want. Moreover, be it for rules or configurations, ACML can manage random seed and transparently save/load them to allow reproducibility of experiments.

For things to work as you expect, observe the following discipline:

1. use functions from the API for (pseudo-)random generation
   - `ca_random` when it concerns the rule,
   - `conf_random` when it concerns the configuration;
2. specify whether you want a deterministic or a stochastic CA.

```
open Acml

let transform a =
  set_dimension a (param a "dimension" 1 2 2);
  set_cardinality a (param a "states" 2 100 5);
  set_moore_neighbourhood a (param a "radius" 1 50 1);
```

Now we define a local function which uses (pseudo-)randomness: it chooses at random one of the states appearing in the neighbourhood. Since it concerns the local rule, we use `ca_random` from the API to get random numbers.

```
let local_function x = x.(ca_random (Array.length x)) in
```

We must tell ACML what we want (stochastic or deterministic) before registering the local rule. To easily test all possibilities via the user interface, let's define some choice parameter.

```
    (match choice_param a "determinism type" ["I want it stochastic";
                                              "I want it deterministic";
                                             "I foret to specify what I want"] with
         "I want it stochastic" → declare_stochastic a
       | "I want it deterministic" → ensure_determinism a
       | _ → ());
```

Then the function is registered as usual.

```
    set_rule a local_function

let _ = valid transform
```

Let's explain what happens in each situation:

- *I want it stochastic*: you get always the same stochastic CA;
- *I want it deterministic*: you get a deterministic CA chosen randomly among 'captive cellular automata' (i.e. CA for which any transition leads to a state which was already present in the neighbourhood);
- *I foret to specify what I want*: by default, ACML considers that the local rule you give is deterministic, and, for efficiency, it can decide to add a cache mecanism; therefore you get a deterministic CA chosen randomly among captive CA when the cache mecanism is added (for small size CA), and the stochastic CA mentioned above when no cache mecanism is added (for large CA).