TREATING AGENT-INTERACTIONS IN ABMS AS ALGEBRAIC STRUCTURES

Gene Callahan

ecallahan2@sjcny.edu

St. Joseph's College, NY

David Seppala-Holtzman

dholtzman@sjcny.edu

St. Joseph's College, NY

1. INTRODUCTION

This paper describes an ongoing effort to treat agent interactions as an abstract algebraic structure. For the time being, the structure we have chosen to use is the module, which is described in detail in the next section of this paper. It is quite possible that, as time goes on, we shall want to make our axioms more restrictive, and assume we are operating on a field, but we will only do this if it proves fruitful. (A module is a generalization of a vector space, in that it operates over a ring, rather than more restrictively, over a field.)

The primary motivations for this attempt it are to increase code reuse, and ultimately to enable the creation of agent-based models through the filling in of forms, choosing among various module operations and chaining them together to produce customized agent behavior. So far, the results have been promising, although there is much work to be done.

2. THE NATURE OF THE STRUCTURE

As noted above, we have initially decided to make our algebraic structure a module. To be a module an algebraic structure must contain a primary set that is an additive group, G, satisfying four group axioms: closure, associativity, identity and invertibility. There is an operator ⊕ which takes two elements of the group and yields a third element and its operation satisfies these axioms.

In addition, a module contains a secondary set, R, a ring of coefficients, with a second operation, ⊗, which takes an element of R and an element of G and produces and element of G.

A. An Analysis of a Generic ABM Interaction as a Module

i. **Elements: Following Whitehead (2014), we call the elements of our group *prehensions*. A prehension can be roughly understood as a state of affairs in the world *as seen from a particular point of view*. (In this case the world is the world of our model [see Morgan 2012], but Whitehead views this as a useful metaphysics for the actual world.)**

ii. **The operation ⊕, which we will call "prehend", accepts two prehensions as arguments and produces a third prehension.**

   1. *Axioms:*

a. ***Closure:*** *Every prehending involving two prehensions will produce a prehension.*

b. ***Associativity:*** *$(a \oplus b) \oplus c = a \oplus (b \oplus c)$*
In a typical agent model, this will mean that we must ensure that, say, a neighborhood can interact with a neighborhood $(b \oplus c)$, and then with an agent $(a \oplus (b \oplus c))$. Furthermore, this must produce an identical prehension to that produced by an agent interacting with one neighborhood and then another one $((a \oplus b) \oplus c)$.

c. ***Identity:*** *Any prehension prehending the null prehension produces itself.*

d. ***Invertibility:*** *For any prehension, there is another prehension that combines with it to produce the null prehension.*

iii. The operation ⊗, which we will call "intensify" (although it may also de-intensify) accepts an element of R and an element of G (a prehension), and produces an element of G.

1. *Axioms:*
   a. *$a, b \in G$:*
      i. $(a \oplus b)x = ax \oplus bx$
      ii. $a(x \oplus y) = ax \oplus ay$

iv. Meaning:

1. *An agent's prehension of itself is its view of its own internal state.*
2. *An agent's prehension of its environment is its view of its surroundings.*
3. *But from the point of view of the prehension module, these prehensions are interchangeable.*
4. *A null prehension could arise, e.g., from the environment when an agent has no neighbors. It could arise internally when an agent has "no opinion" on the relevant parameters, e.g., a color-blind agent in our fashion model.*
5. *Invertibility may occur, for instance, when an agent has some internal tendency to act in some way (e.g., to move to a new neighborhood or switch fashions) but some force in the environment exactly offsets that tendency (e.g., that "authorities" establish come penalty for so acting).*
6. *An intensification of a prehension leaves the elements of the prehension in the same internal relationship, but they are scaled up or down relative to other prehensions. This is useful for capturing situations like the gradual dissipation of an attitude, or increasing fanaticism over time.*

3. THE ADVANTAGES OF EMPLOYING THIS ABSTRACTION

A. Why bother?

i. We achieve a uniform template for all models as far as how agents interact with their environment.

Programming new models becomes much easier. We have, in fact, been achieving code reductions on the order of 30-40% for models we have converted to the new paradigm.

ii. We will have taken a huge step towards enabling "fill-in-the-template" style programming of ABMs.

A non-programming user ultimately should be able to build his model by choosing from a palette of possible module operations, setting parameters for each, and chaining them together.

iii. We open up the possibility of using known properties of modules to identify properties of our ABM.

This has not been achieved yet, but the possibility remains open.

4. A SKETCH OF THE USUAL ACTION PATTERN

An agent gathers together a prehension of its environment, and then combines that with how it views its own state (its self-prehension) to produce a new prehension. The new prehension may simply be adopted, or it may trigger some further step, such as a movement in space on the part of the agent. And each agent may prehend not only a local slice of the overall environment, but a different sized slice as well.

The combination of very generic processing in treating the prehensions with individual agent flexibility in responding to that outcome allows us to combine some of the best features of older-style simulations with ABMs.

A. Some examples:

i. Fashion model

The prehension returned from the environment will be a vector representing the mix of fashions being worn in the agent's neighborhood. The agent will modify her own preferred fashion based on that vector and the sort of agent she is. (Fashion followers will incline toward the prevailing fashion, while trend-setters will move away from it.)

ii. Financial market model

The prehension returned from the environment will be a vector representing the buy/sell sentiment prevailing around the agent. The agent will modify her own market stance based on that vector and the sort of agent she is. (Chart followers will incline toward the prevailing sentiment, while value investors will move away from it.)

iii. Schelling's segregation model

In this model, the agent prehends the color mix of his neighborhood. We normalize the vector we get back, and project it onto the axis representing his own color. His self-prehension represents his minimum percentage "like him" he will tolerate in his neighborhood. If the normalized, projected vector representing the neighborhood falls below that amount, the agent jumps to a random cell.

However, while the above may be typical, our system allows for the reverse: in some models (e.g., Forest Fire), it may be the environment that adopts the new prehension. Furthermore, environmental prehensions may interact directly with each other as well.

5. CONCLUSION

We intend to proceed by next implementing the common forest fire model as a module, then attempting an abelian sand pile. Then we will proceed to models that much less obviously obviously when themselves to this sort of treatment, such as models of exchange, and the Mengerian emergence of money (Menger, 1892). If these models when themselves to this sort of treatment described here, then our chances of producing a template-driven modeling system appear good indeed.

In any case, even the limited amount of work performed so far offers grounds for hope: we have achieved a great deal of code reuse already, and significantly reduced the size of each of the models we have converted to this paradigm.

6. APPENDICES: IMPLEMENTATION DETAILS

The following are the main modules implementing this paradigm. They have been written up for maximum readability, in both the comments and especially the code itself. (See Knuth, 1992.)

A. prehension.py

```
1. """
2. prehension.py
3. The way agents interact is through prehensions.
4. entity.py currently has another notion of prehension: that one
5. and this one must be combined in the future.
6. The base implementation of a prehension is as a vector.
7. As of the moment, this is essentially a wrapper around numpy's vector
8. functions. But the reason to do it that way is because other models might
```

```
9.  want a prehension that is NOT a vector.
10. Sub-class this to instantiate another implementation.
11. """
12.
13. import math
14. import numpy as np
15. # import logging
16.
17. # x and y indices
18. X = 0
19. Y = 1
20.
21. # Set up constants for some common vectors: this will save time and memor
    y.
22. X_VEC = np.array([1, 0])
23. Y_VEC = np.array([0, 1])
24. NULL_VEC = np.array([0, 0])
25. NEUT_VEC = np.array([.7071068, .7071068])
26.
27.
28. class Prehension:
29.     """
30.     This duplicates the beginning of this paper!
31.     """
32. # we must pre-
    declare these, then use them, then init them at the bottom
33. #  of the file.
34.     X_PRE = None
35.     Y_PRE = None
36.     NULL_PRE = None
37.     NEUT_PRE = None
38.
39.     @classmethod
40.     def from_vector(cls, v):
41.         """
42.         Convenience method to turn a vector into a prehension.
43.         """
44.         p = Prehension()
45.         p.vector = v
46.         return p
47.
48.     def __init__(self, x=0, y=0):
49.         self.vector = np.array([x, y])
50.
51.     def __str__(self):
52.         return ("x: %f, y: %f" % (self.vector[X], self.vector[Y]))
53.
54.     def prehend(self, other):
55.         """
56.         In this class, a prehension prehends another prehension
57.         through vector addition.
58.         other: prehension to prehend
59.         """
60.         return Prehension.from_vector(self.vector + other.vector)
61.
62.     def intensify(self, a):
63.         """
64.         Here this is scalar multiplication of a vector.
65.         a: scalar to multiply by.
66.         """
67.         return Prehension.from_vector(self.vector * a)
68.
69.     def direction(self):
70.         """
71.         This gets us the orientation of the vector: x, y, or neutral.
```

```
        We use it, for instance, to set an agent's market stance to
        buy or sell.
        """
        if self.vector[X] > self.vector[Y]:
            return Prehension.X_PRE
        elif self.vector[X] < self.vector[Y]:
            return Prehension.Y_PRE
        else:
            return Prehension.NEUT_PRE

    def project(self, x_or_y):
        """
        Projects the vector onto the x or y axis.
        Pass in X or Y as declared above.
        """
        return self.vector[x_or_y]

    def equals(self, other):
        """
        For prehensions of the base type, they are equal
        when their vetors are equal.
        """
        return np.array_equal(self.vector, other.vector)

    def reverse(self):
        """
        Reverse the vector.
        """
               new_vec = np.array(np.flipud(self.vector))
               return Prehension.from_vector(new_vec)

           def normalize(self):
               """
               Return a normalized prehension.
               If we get the NULL prehension, just return it.
               """
               if self.equals(Prehension.NULL_PRE):
                   return Prehension.NULL_PRE

               return Prehension.from_vector(self.vector
                         / np.linalg.norm(self.vector))


      # Now we actually initialize the prehensions we declared above.
      #  This can't be done earlier, since Prehension was just defined.
      Prehension.X_PRE = Prehension.from_vector(X_VEC)
      Prehension.Y_PRE = Prehension.from_vector(Y_VEC)
      Prehension.NULL_PRE = Prehension.from_vector(NULL_VEC)
      Prehension.NEUT_PRE = Prehension.from_vector(NEUT_VEC)


      def stance_pct_to_pre(pct, x_or_y):
          """
          pct is our % of the way to the y-axis from
          the x-axis around the unit circle.
          (If x_or_y == Y, it is the opposite.)
          It will return the x, y coordinates of
          the point that % of the way.
          I.e., .5 returns NEUT_VEC, 0 returns X_VEC.
          """
          if x_or_y == Y:
              pct = 1 - pct
          if pct == 0:
              return Prehension.X_PRE
          elif pct == .5:
```

```
            return Prehension.NEUT_PRE
        elif pct == 1:
            return Prehension.Y_PRE
        else:
            angle = 90 * pct
            x = math.cos(math.radians(angle))
            y = math.sin(math.radians(angle))
            return Prehension(x, y)
```

B. prehension_agent.py

```
    """
    prehension_agent.py
    An agent that use Prehensions to understand its environment.

    """
    import indra.grid_agent as ga
    import indra.prehension as pre


    class PrehensionAgent(ga.GridAgent):
        """
        An agent that use Prehensions to understand its environment.
        """
        def __init__(self, name, goal, max_move=1, max_detect=1):
            super().__init__(name, goal, max_move, max_detect)
            self.my_filter = None
            self.stance = pre.Prehension()

        def survey_env(self):
            """
            Look around and see what surrounds us.
            """
            super().survey_env()
            other_pre = pre.Prehension()
            for other in self.neighbor_iter(view=self.my_view,
                    filt_func=self.my_filter):
                # accumulate prehensions:
                other_pre = other.visible_stance().prehend(other_pre)
            return other_pre

        def visible_stance(self):
            """
            By default, just our stance. But we may override this.
            For instance, perhaps just our direction is visible.
            """
            return self.stance
```

C. segregation_model.py

```
    import random
    import indra.prehension as pre
    import indra.prehension_agent as pa
    import indra.grid_env as grid

    MOVE = True
    STAY = False
    RED = pre.X
    BLUE = pre.Y
    RED_PRE = pre.Prehension.X_PRE
```

```
190.      BLUE_PRE = pre.Prehension.Y_PRE
191.      RED_AGENT = "RedAgent"
192.      BLUE_AGENT = "BlueAgent"
193.      AGENT_TYPES = {RED: RED_AGENT, BLUE: BLUE_AGENT}
194.
195.
196.      class SegregationAgent(pa.PrehensionAgent):
197.          """
198.          An agent that moves location based on its neighbors' types
199.          """
200.          def __init__(self, name, goal, min_tol, max_tol, max_move=100,
201.                       max_detect=1):
202.              super().__init__(name, goal, max_move=max_move,
203.              max_detect=max_detect)
204.              self.tolerance = random.uniform(max_tol, min_tol)
205.              self.stance = None
206.              self.orientation = None
207.              self.visible_pre = None
208.
209.          def eval_env(self, other_pre):
210.              """
211.              Use the results of surveying the env to decide what to do.
212.              """
213.              # no neighbors, we stay put:
214.              if other_pre.equals(pre.Prehension.NULL_PRE):
215.                  return STAY
216.
217.              # otherwise, see how we like the hood
218.              other_pre = other_pre.normalize()
219.              other_projection = other_pre.project(self.orientation)
220.              my_projection = self.stance.project(self.orientation)
221.              if other_projection < my_projection:
222.                  return MOVE
223.              else:
224.                  return STAY
225.
226.          def respond_to_cond(self, env_vars=None):
227.              """
228.              If we don't like the neighborhood, jump to an empty cell.
229.              """
230.              self.move_to_empty()
231.
232.          def visible_stance(self):
233.              """
234.              Our visible stance differs from our internal one.
235.              It is just our "color."
236.              """
237.              return self.visible_pre
238.
239.
240.      class BlueAgent(SegregationAgent):
241.          """
242.          We set our stance.
243.          """
244.          def __init__(self, name, goal, min_tol, max_tol, max_move=100,
245.                       max_detect=1):
246.              super().__init__(name, goal, min_tol, max_tol,
247.              max_move=max_move, max_detect=max_detect)
248.              self.orientation = BLUE
249.              self.visible_pre = BLUE_PRE
250.              self.stance = pre.stance_pct_to_pre(self.tolerance, BLUE)
251.
252.
253.      class RedAgent(SegregationAgent):
254.          """
```

```
255.          We set our stance.
256.          """
257.          def __init__(self, name, goal, min_tol, max_tol, max_move=100,
258.                       max_detect=1):
259.              super().__init__(name, goal, min_tol, max_tol,
260.                               max_move=max_move, max_detect=max_detect)
261.              self.orientation = RED
262.              self.visible_pre = RED_PRE
263.              self.stance = pre.stance_pct_to_pre(self.tolerance, RED)
264.
265.
266.
267.      class SegregationEnv(grid.GridEnv):
268.          """
269.          The segregation model environment, concerned with bookkeeping.
270.          """
271.
272.          def __init__(self, name, width, height, torus=False,
273.      model_nm="Segregation"):
274.
275.              super().__init__(name, width, height, torus=False,
276.                               model_nm=model_nm)
277.              self.plot_title = name
278.              # setting our colors adds varieties as well!
279.              self.set_var_color(AGENT_TYPES[BLUE], 'b')
280.              self.set_var_color(AGENT_TYPES[RED], 'r')
281.              self.num_moves = 0
282.              self.move_hist = []
283.
284.          def move_to_empty(self, agent, grid_view=None):
285.              super().move_to_empty(agent, grid_view)
286.              self.num_moves += 1
287.
288.          def census(self, disp=True):
289.              """
290.              Take a census recording the number of moves.
291.              """
292.              self.move_hist.append(self.num_moves)
293.              self.user.tell("Moves per turn: " + str(self.move_hist))
294.              self.num_moves = 0
```

D. fashion_model.py

```
295.      """
296.      fashion_model.py
297.      A fashion model that includes followers and hipsters

298.      changing fashions based on each other's choices.
299.      """
300.      # import logging
301.      import indra.display_methods as disp
302.      import indra.menu as menu
303.      import indra.two_pop_model as tp
304.
305.
306.      class Follower(tp.Follower):
307.          """
308.          A fashion follower: tries to switch to hipsters' fashions.
309.          """
```

```
310.          def __init__(self, name, goal, max_move, variability=.5):
311.              super().__init__(name, goal, max_move, variability)
312.              self.other = Hipster
313.
314.
315.      class Hipster(tp.Leader):
316.          """
317.          A fashion hipster: tries to not look like followers.

318.          """
319.          def __init__(self, name, goal, max_move, variability=.5):
320.              super().__init__(name, goal, max_move, variability)
321.              self.other = Follower
322.
323.
324.      class Society(tp.TwoPopEnv):
325.          """
326.          A society of hipsters and followers.
327.          """
328.          def __init__(self, name, length, height, model_nm=None,
329.      torus=False):
330.              super().__init__(name, length, height, model_nm=model_nm,
331.              torus=False, postact=True)
332.              self.stances = ["blue", "red"]
333.              self.line_graph_title = "A. Smith's fashion model"
334.              self.set_var_color('Hipster', disp.GREEN)
335.              self.set_var_color('Follower', disp.MAGENTA)
336.              self.menu.view.add_menu_item("v",
337.                  menu.MenuLeaf("(v)iew fashions",
338.                  self.view_pop))
```

E. fmarket_model.py

```
339.      """
340.      fmarket_model.py
341.      A financial market model that includes chart followers

342.      and value investors.
343.      """
344.      import logging
345.      import indra.menu as menu
346.      import indra.display_methods as disp
347.      import indra.two_pop_model as tp
348.
349.      INIT_PRICE = 10.0
350.      INIT_ENDOW = INIT_PRICE * 10.0
351.      BUY = tp.INIT_FLWR
352.      SELL = tp.INIT_LEDR
353.
354.
355.      class FinancialAgent(tp.TwoPopAgent):
356.          """
357.          A financial agent who trades an asset.
358.          """
359.          def __init__(self, name, goal, max_move, variability=.5):
360.              super().__init__(name, goal, max_move, variability)
361.              self.profit = 0.0
362.              self.funds = INIT_ENDOW
363.
```

```
364.          def direction_changed(self, curr_direct, new_direct):

365.              if not new_direct.equals(curr_direct):
366.                  if new_direct.equals(BUY):
367.                      self.funds -= self.env.asset_price
368.                  else:
369.                      self.funds += self.env.asset_price
370.                  self.calc_profit()
371.
372.          def calc_profit(self):
373.              self.profit = self.funds - INIT_ENDOW
374.              self.env.record_profit(self, self.profit)
375.
376.          def add_env(self, env):
377.              super().add_env(env)
378.              if self.stance == BUY:
379.                  self.funds -= self.env.asset_price
380.                  self.calc_profit()
381.
382.
383.      class ChartFollower(FinancialAgent, tp.Follower):
384.          """
385.          A trend follower: buys what others are buying.
386.          """
387.          def __init__(self, name, goal, max_move, variability=.5):
388.              super().__init__(name, goal, max_move, variability)
389.              self.other = ValueInvestor
390.
391.
392.      class ValueInvestor(FinancialAgent, tp.Leader):
393.          """
394.          A value investor: buys what others are not buying.
395.          """
396.          def __init__(self, name, goal, max_move, variability=.5):
397.              super().__init__(name, goal, max_move, variability)
398.              self.other = ChartFollower
399.
400.      class FinMarket(tp.TwoPopEnv):
401.          """
402.          A society of value investors and chart followers.
403.          """
404.          def __init__(self, name, length, height, model_nm=None,
405.                       torus=False):
406.              super().__init__(name, length, height, model_nm=model_nm,
407.              torus=False, postact=True)
408.              self.total_pop = 0  # to be set once we add agents
409.              self.asset_price = INIT_PRICE  # arbitrary starting point
410.              self.price_hist = []
411.              self.max_abs_pmove = .5
412.              self.stances = ["buy", "sell"]
413.              self.menu.view.add_menu_item("v",
414.         menu.MenuLeaf("(v)iew asset price",
415.      self.view_price))
416.              self.follower_profit = 0.0
417.              self.value_profit = 0.0
418.
419.          def record_profit(self, agent, profit):
420.              if isinstance(agent, ChartFollower):
421.                  self.follower_profit += profit
422.              else:
423.                  self.value_profit += profit
```

7. BIBLIOGRAPHY

Knuth, Donald E. 1992. *Literate Programming*. Stanford, California: Center for the Study of Language and Information.

Menger, Carl. 1892. "On the Origin of Money." Accessed March 15, 2015. http://www.monadnock.net/menger/money.html.

Morgan, Mary S. 2012. *The World in the Model: How Economists Work and Think*. Cambridge; New York: Cambridge University Press.

Schelling, Thomas C. 2006. *Micromotives and Macrobehavior*. New York: Norton.

Stepanov, Alexander A, and Daniel E Rose. 2015. *From Mathematics to Generic Programming*. Upper Saddle River, NJ [u.a.]: Addison-Wesley.

Whitehead, Alfred North. 2014. *Process and Reality*. [S.l.]: Free Press. http://rbdigital.oneclickdigital.com.