Formalizing Utility Semantics With Infinite Group Generators

by Sven Nilsen, 2018

In this paper I outline an overall idea for fixing semantics of artificial intelligent agent goals: By taking advantage of advanced techniques of abstract algebra, one can define precisely the language used to describe goals without specifying the implementation of that language. By computing utility from objects generated by infinite group generators, this guarantees that semantics is preserved under modification by generators. I demonstrate this technique by defining a universal problem solver.

Think of a generator as a way to modify an object. The way an object gets modified has certain mathematical properties which are defined as the concept of a group (or groupoid) in mathematics.

To explain how this works, I will give a simple description of a universal problem solver based on natural numbers and evolution.

You run an evolutionary algorithm on a population of artificial agents that each are described with a genetic code. The genetic code takes the form of a binary representation of natural numbers. At each generation, the agents are evaluated by a utility/fitness function that takes the genetic code as input and returns a score of how well the agent solved the task:

$$U : \text{binary\_code} \rightarrow \text{real}$$

The utility function makes its decisions in the language of infinite group generators. The generators are defined as the following higher order generator:

$$\text{gen} : \text{nat} \times \text{binary\_code} \times \text{binary\_code} \rightarrow \text{binary\_code}$$
$$\text{gen} := \backslash(n, a, b) = a + \text{if bit}(b, n)\ \{\ 2^n\ \}\ \text{else}\ \{\ 0\ \}$$

$$\text{bit} : \text{binary\_code} \times \text{nat} \rightarrow \text{bool}$$

Inverse generators:

$$\text{gen}^{-1} := \backslash(n, a, b) = a - \text{bit}(b, n)\ \{\ 2^n\ \}\ \text{else}\ \{\ 0\ \}$$

Identity element (this is the same for every generator):

$$e = 0$$

The algorithm for evolving the population can be arbitrary, but to keep all possible agents reachable there must exists one agent `x : (bit 0)`. This makes the algorithm universal for problem solving.

When evolving, one agent `a` functions as a blueprint and another agent `b` as the source of new genes and mutations. A single bit is taken from the agent `b` and inserted into `a` that either puts the bit there or cause upward mutation. The reverse operation removes the bit or cause downward mutation.

To make the utility function `U` sound, one must use the language of an infinite amount of group generators to specify the goal. It is not permitted to use other operations, unless they are definable in the same language. However, it is possible to constrain the generators for narrow problem solving.

This language is of a such kind that all operations are possible to define, thanks to the construction of the identity element `e` and selecting a proper generator.

Example of things that can be expressed in this language:

$a +_n b = \text{gen}(n, a, b)$	Insert gene
$a -_n b = \text{gen}^{-1}(n, a, b)$	Remove gene
$\forall\ n \{ a +_n e = a \}$	For any `$+_n$`, using `e` as source for genes has no effect
$\forall\ n \{ a -_n e = a \}$	For any `$-_n$`, using `e` as source for genes has no effect
$e +_0 a$	Agent `a`'s bit at location 0
$e +_0 a = e$	Agent `a`'s bit at location 0 has value 0
$b_{min} <= a <= b_{max}$	`$b_{min}$` and `$b_{max}$` defines an AABB region of agents
$e +_3 a < e +_3 b$	`a`'s 3rd bit is less than `b`'s third bit
$\forall\ i\ [3, 5) \{ e +_i a = e +_i b \}$	Bits `3` and `4` are the same in `a` and `b`

Comparison operators up to some finite limit can be defined if you can compare elements for equality. These are the boolean equivalent operators. There are more than one way to define comparisons, some are more impractical than others. For example, addition on real numbers is uncomputable since you order element `$a < b$` when `$\forall\ n \{ a +_n b = a \}$` and this forms a category of partially ordered objects. In practice you just use the normal definitions and keep the mathematical definitions only for theorems.

In practice, this language is powerful enough so you can read any part of the binary code and perform any kind of computation. So, since binary code can represent any object, why not just use binary code?

The idea is that there are many ways to define generators. Some are more efficient at specific problems than others. For example, you can create a cyclic generator that does not cause mutations. Or, you can create a complex definition of a concept that is defined formally in this language. While some structures are universal, there are others that are much more restricted. Generic algorithms can be written to work with any structure. Mathematical theorems can be proven using a common definition.

Modifications are only allowed to happen in a certain way such that if you keep track of all the changes, one can always undo them later. Here is an example showing how changes are reversible:

$$1011 -_2 0100 = 0111 \qquad 1011 -_2 0000 = 1011$$
$$0111 +_2 0100 = 1011 \qquad 1011 +_2 0000 = 1011$$

A curious property in this case: As the diversity of genes increases in a growing population of agents, new generators are unlocked by mutation of higher bits, speeding up evolution, even they all can be generated from a single agent and the identity element.