SLiM Workshop Series #12: Callbacks III: `modifyChild()`

Copyright © 2020-2021 Benjamin C. Haller. All rights reserved.

1. Start a new model in SLiMgui

Run SLiMgui, and close any existing model windows. Press command-N to make a new model window. Adapt the default script to the following code, or create this script from scratch:

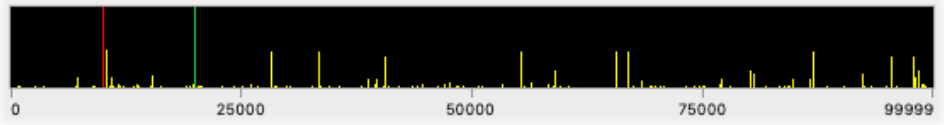
```
initialize() {
   initializeMutationRate(1e-7);
   initializeMutationType("m1", 0.5, "f", 0.0);

   initializeMutationType("m2", 0.5, "f", 0.5);  // mutation A
   m2.convertToSubstitution = F;
   m2.color = "red";

   initializeMutationType("m3", 0.5, "f", 0.5);  // mutation B
   m3.convertToSubstitution = F;
   m3.color = "#20D033";

   initializeGenomicElementType("g1", m1, 1.0);
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}
1 late() {
   sim.addSubpop("p1", 500);
   sample(p1.genomes, 20).addNewDrawnMutation(m2, 10000);  // add A
   sample(p1.genomes, 20).addNewDrawnMutation(m3, 20000);  // add B
}
10000 { sim.simulationFinished(); }
```

This is a fairly simple one-subpopulation model with a chromosome that undergoes only neutral (`m1`) mutations. However, two other mutation types are defined, `m2` and `m3`, both of which are strongly beneficial (s=0.5). The model adds a single `m2` mutation (called "A") to `20` randomly chosen genomes, and a single `m3` mutation (called "B") to `20` randomly chosen genomes. It runs until generation `10000` and terminates.

We set two properties on those mutation types that we haven't encountered before: `convertToSubstitution` and `color`. The `convertToSubstitution` property, if set to `F`, prevents SLiM from replacing mutations of that mutation type with `Substitution` objects when they fix; the mutations will just continue segregating in the population, at a frequency of `1.0`, forever. This should be used with caution, since if large numbers of mutations are prevented from being substituted SLiM will run very slowly (since it will have to manage all of those mutations during offspring generation, fitness calculation, etc.). Here, however, we use it for the A and B mutations because we want them to interact *epistatically* – we want the fitness effect of A to depend upon whether B is also present, and *vice versa*. This epistatic interaction continues to happen even if either A or B fixes, so for simplicity we prevent them from being substituted when they fix. Neutral mutations will continue to be substituted upon fixation as usual, since we didn't change `m1`'s default `convertToSubstitution` value of `T`.

The `color` property sets the color used when displaying mutations of a given mutation type in SLiMgui (it has no effect upon models run at the command line). We set `m2` to a color of `"red"`, which, unsurprisingly, means it will display as red. SLiM knows many named colors; in fact it uses the named color list from R, so those familiar with R colors will be at home. As in R, colors can also be specified with a hexadecimal RGB color; the color `"#20D033"` is a shade of green. We won't get into the weeds of hexadecimal, RGB, etc., at this point; the Eidos manual covers this topic.

If we run this model, the A and B mutations will almost always fix (since they are strongly beneficial and were introduced into 20 genomes initially), and we'll see a picture like this:



The A and B mutations, in red and green, can be clearly seen toward the left edge of the chromosome; both have fixed, but they have not been converted to `Substitution` objects because their mutation types have a `convertToSubstitution` value of `F`.

2. Add dominant lethal epistasis with a `modifyChild()` callback

We would like A and B to exhibit dominant lethal epistasis: any offspring that contains both mutations (whether homozygously or heterozygously) should experience developmental problems that prevent it from surviving. With a `modifyChild()` callback this is very easy to implement. This callback type is called once for each focal proposed offspring: every time that SLiM has generated an offspring individual and is about to add it to the target subpopulation, it calls any applicable `modifyChild()` callbacks to allow them to modify or veto the proposed child. Since we want to model a lethal developmental problem, we will veto proposed children that carry both A and B. Add the following code to your script (anywhere – at the end is fine):

```
modifyChild() {
   hasMutA = any(child.genomes.countOfMutationsOfType(m2) > 0);
   hasMutB = any(child.genomes.countOfMutationsOfType(m3) > 0);
   if (hasMutA & hasMutB)
      return F;
   return T;
}
```

First the callback determines whether the proposed child's genomes contain any `m2` or `m3` mutations. These tests are written in a general way, so that the code works even if multiple `m2` or `m3` mutations could exist at different genome positions; if the model can be hard-coded for a single `m2` and `m3` mutation at known positions, the `containsMarkerMutation()` method could be used instead and would be faster. We'll keep it general here, though. It is probably obvious, but the proposed child was passed to the callback through pseudo-parameter `child`.

The rest of the callback is exceedingly simple. If the proposed child contains both A and B mutations in its genomes, it returns `F`; this tells SLiM to suppress the generation of this proposed child. Otherwise, it returns `T`, telling SLiM to proceed with generation normally.

Recycle and run this model several times, and notice that in every run you get either the A mutation or the B mutation, but never both. The lethal epistasis completely prevents co-existence.

EXERCISE: Switch the callback to implement *recessive* lethal epistasis. In this case, the proposed child should be suppressed only if it is homozygous for both A and B. Homozygosity can be determined from the number of `m2` or `m3` mutations present in the proposed child's genomes.

In this exercise we're looking only at *lethal* epistasis, since a `modifyChild()` callback is well-suited to implementing that. It is straightforward to model non-lethal epistasis too, using a similar sort of strategy in a `fitness()` callback. The SLiM manual has an example of this type of model.

3. Cultural inheritance

So far we have looked at a `modifyChild()` callback that returns `F` to veto a proposed child. It is also common to use `modifyChild()` callbacks to, as the name suggests, *modify* a proposed child without

vetoing it. We'll shift gears now completely, to look at such a model: a model of non-genetic cultural inheritance of a trait.

This model won't run until it's all built, but let's nevertheless build it one block at a time. The first component is the `initialize()` callback. Close any open SLiMgui windows, and then make a new window and modify its script to look like this:

```
initialize() {
   initializeMutationRate(1e-7);
   initializeMutationType("m1", 0.5, "f", 0.0);   // neutral
   initializeMutationType("m2", 0.5, "f", 0.1);   // lactase-promoting
   m2.convertToSubstitution = F;
   initializeGenomicElementType("g1", c(m1,m2), c(0.99,0.01));
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}
```

There's a relatively infrequent mutation type, `m2`, that has a somewhat beneficial effect, and is prevented from being substituted at fixation using `convertToSubstitution` just as before. This mutation type promotes production of lactase (the enzyme that allows lactose, the common sugar in milk, to be digested). Next, create a new subpopulation:

```
1 {
   sim.addSubpop("p1", 1000);

   // start as mostly non-milk-drinkers
   p1.individuals.tag = rbinom(1000, 1, 0.01);
}
```

We use `addSubpop()` as usual. Then we do a vectorized property assignment into the `tag` property of all of the `p1` individuals (`tag` is like the `tagF` property we saw before, but of type `integer`). The values assigned come from `rbinom()`, which generates draws from a binomial distribution. In this case, the values will be `0` about 99% of the time, representing individuals who – for cultural reasons – do not drink milk as adults, whereas about 1% of values will be `1`, representing individuals who culturally *do* drink milk as adults. You can use the online help to read about `rbinom()`.

Next comes a `late()` event that runs in every generation, for purposes of output:

```
late() {
   // dark blue for non-milk-drinkers, light gray for milk-drinkers
   inds = p1.individuals;
   inds.color = ifelse(inds.tag == 0, "darkblue", "lightgray");
   catn(sim.generation + ": " + mean(inds.tag));
}
```

This assigns color values to individuals in each generation, based upon their `tag` value; if they are a non-milk-drinker (`tag` of `0`) they are colored dark blue, whereas if they are a milk-drinker (`tag` `1`) they are colored light gray. We previously saw the use of `mutationType`'s `color` property; this is essentially the same, and again these named colors are part of the standard R named color list. The vector of color names assigned into `inds.color` comes from the `ifelse()` function, which is essentially a vectorized `if-else` statement; see its documentation for further details. Finally, the event prints the fraction of milk-drinkers in each generation.

Next, add a very simple `fitness()` callback:

```
fitness(m2) {
   // deleterious for non-milk-drinkers, beneficial for milk-drinkers
   return (individual.tag == 0) ? 0.95 else relFitness;
}
```

This makes the `m2` mutations slightly deleterious (with complete dominance) when found in non-milk-drinkers, by returning `0.95`. In milk-drinkers, on the other hand, it returns `relFitness` to use SLiM's default fitness effect for the mutation, which was set up to be somewhat beneficial. Since we've covered `fitness()` callbacks already, this step is probably clear.

Define an end to the simulation with a termination event:

```
10000 { sim.simulationFinished(); }
```

And now, finally, we get to the interesting bit: using a `modifyChild()` callback to provide non-genetic inheritance of the cultural preference for milk-drinking. Add this to the script:

```
modifyChild() {
   // inherit culture from parents, with some deviation
   parentCulture = (parent1.tag + parent2.tag) / 2;
   childCulture = rbinom(1, 1, 0.01 + 0.98 * parentCulture);
   child.tag = childCulture;
   return T;
}
```

The parental values for the milk-drinking trait are obtained from their `tag` values (`parent1` and `parent2` are pseudo-parameters, of course, representing the first and second parents of the proposed child). Those values are averaged, and then `rbinom()` is used to draw a milk-drinking trait value for the child that is very heavily based upon the culture of its parents, but contains a tiny random element to provide a bit of stochasticity in cultural transmission. The child's chosen preference is recorded in its `tag` value, and finally `T` is returned to tell SLiM to proceed with generation of the proposed child.

With that, all the script blocks are in place and we can run the model. You can visualize the shift from non-milk-drinking to milk-drinking in SLiMgui, through the coloring of individuals. The individuals rapidly shift from blue to gray, and the output from `catn()` shows the same pattern:

```
1: 0.021
2: 0.029
3: 0.032
4: 0.048
5: 0.051
...
100: 0.429

500: 0.956
...
1000: 0.984
...
```

This outcome is reproducible across runs. A question to briefly ponder: why does it happen so reliably? After all, the population starts out as almost entirely non-milk-drinkers, and the lactase-promoting mutations are deleterious in non-milk-drinkers. Shouldn't this mean that lactase-promoting mutations are weeded out, and therefore that there is no advantage to milk-drinking?

EXERCISE: Let's explore this model a bit. Do your **BEEP!** first, now. Then, one thing to look at, in understanding the behavior of this model, is what the model does if the mutation rate is simply set to zero. What do you observe? Why does it happen? For another clue, watch the dynamics of the model (with the mutation rate restored to `1e-7`) as it runs. Does it transition from low to high milk-drinking in a linear fashion?

4. BEEP! With extra time, look at SLiM manual section 12.3, a very interesting application of a `modifyChild()` callback to the implementation of a CRISPR gene drive.