not record the addition and removal of spurious mutations, and all of the second genomes of individuals in this model will be recorded as being null genomes (which you can test for with `pyslim`), producing a cleaner recorded tree sequence that better reflects the fact that those second genomes do not actually exist at all, conceptually. The third benefit of using `addRecombinant()` is that it allows more complex modes of offspring generation to be expressed as well; as an example, in the next section we will see how to model horizontal gene transfer in bacteria using `addRecombinant()` to express the horizontal gene transfer to SLiM.

16.14 Modeling clonal haploid bacteria with horizontal gene transfer

In section 16.13 we looked at a model of clonal haploids using `addRecombinant()`, as an alternative to the original haploid clonal model presented in section 14.9. The use of `addRecombinant()` allowed the details of child generation to be expressed precisely to SLiM, facilitating a simpler model design and more accurate recording of ancestry in the tree sequence (if tree-sequence recording were enabled; see section 1.7). In this section we'll explore those benefits in more detail in a model of horizontal gene transfer in bacteria.

This is a more complex model than that of section 16.13, so let's take things in two steps. First, here is all of the code except the `reproduction()` callback:

```
initialize() {
    initializeSLiMModelType("nonWF");
    defineConstant("K", 1e5);                       // carrying capacity
    defineConstant("L", 1e5);                       // chromosome length
    defineConstant("H", 0.001);                     // HGT probability
    initializeMutationType("m1", 1.0, "f", 0.0);    // neutral (unused)
    initializeMutationType("m2", 1.0, "f", 0.1);    // beneficial
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, L-1);
    initializeMutationRate(0);                      // no mutations
    initializeRecombinationRate(0);                 // no recombination
}
1 early() {
    // start from two bacteria with different beneficial mutations
    sim.addSubpop("p1", 2, haploid=T);

    // add beneficial mutations to each bacterium, but at different loci
    g = p1.individuals.genome1;
    g[0].addNewDrawnMutation(m2, asInteger(L * 0.25));
    g[1].addNewDrawnMutation(m2, asInteger(L * 0.75));
}
early() {
    // density-dependent population regulation
    p1.fitnessScaling = K / p1.individualCount;
}
late() {
    // detect fixation/loss of the beneficial mutations
    muts = sim.mutations;
    freqs = sim.mutationFrequencies(NULL, muts);

    if (all(freqs == 1.0))
    {
       catn(sim.generation + ": " + sum(freqs == 1.0) + " fixed.");
       sim.simulationFinished();
    }
}
1e6 late() { catn(sim.generation + ": no result."); }
```

The `initialize()` code defines a few constants: the carrying capacity, the chromosome length, and the probability that horizontal gene transfer will occur during a given mitosis event. Note that we will model horizontal gene transfer as occurring during reproduction, rather than as a discrete event occurring later in a bacterium's lifetime. This design is much simpler, particularly if tree-sequence recording is enabled; horizontal gene transfer changes the genealogical relationships among individuals, and tree-sequence recording is not designed to accommodate such changes in the middle of an individual's lifespan. The approximation seems unlikely to matter.

Note also that although we define a neutral mutation type here, `m1`, we do not model neutral mutations, and indeed, we use a mutation rate of `0.0`. This is because a model of this sort is likely to use tree-sequence recording to overlay neutral mutations after the fact for much greater speed; since we haven't gotten into tree-sequence recording yet, however, we will defer that topic until sections 17.1 and 17.2. For now, it suffices to say that we do not model neutral mutations here.

The initial population here consists of just two bacteria, which are set up to carry different beneficial mutations at different locations in the genome. The population will expand exponentially until reaching the carrying capacity of `1e5`. We have used a fairly large population size since we are modeling bacteria, but a carrying capacity of `1e6` or even higher might be desirable for some purposes. Apart from taking more time and memory, this model should scale up without difficulties; the fact that neutral mutations are not included makes it scale much better.

In the `late()` event we detect the fixation or loss of the beneficial mutations; if both mutations have fixed or been lost, the model prints a message indicating how many mutations fixed, and then stops. If the model runs for `1e6` generations without fixation or loss, it stops with a message.

All of that is routine. Now here's the `reproduction()` callback, where the interesting action is:

```
reproduction() {
    if (runif(1) < H)
    {
       // horizontal gene transfer from a randomly chosen individual
       HGTsource = p1.sampleIndividuals(1, exclude=individual).genome1;

       // draw two distinct locations; redraw if we get a duplicate
       do breaks = rdunif(2, max=L-1);
       while (breaks[0] == breaks[1]);

       // HGT from breaks[0] forward to breaks[1] on a circular chromosome
       if (breaks[0] > breaks[1])
          breaks = c(0, breaks[1], breaks[0]);

       subpop.addRecombinant(genome1, HGTsource, breaks, NULL, NULL, NULL);
    }
    else
    {
       // no horizontal gene transfer; clonal replication
       subpop.addRecombinant(genome1, NULL, NULL, NULL, NULL, NULL);
    }
}
```

Each bacterium reproduces exactly once each generation, producing two bacteria from one, which makes sense from the perspective of reproduction by mitosis. This reproduction can happen in two different ways, depending upon a random draw from `runif()`. If the draw is greater than or equal to `H`, reproduction is purely clonal as in the model of section 16.13; that is the `else` clause here. If the draw is less than `H`, horizontal gene transfer occurs, which needs some explanation.

In that case, we first draw a random individual (other than the focal individual) to act as the source for the transfer, and get its first genome. Next we use a `do–while` loop to draw two distinct

locations along the genome; these will be the endpoints of the transfer. Specifically, the transfer will start at `breaks[0]` and go forward to `breaks[1]`. For a bit of extra biological realism, we will model a circular chromosome here, so if `breaks[0]` is greater than `breaks[1]` the transfer will wrap around from the end of the genome to the start, as modelled in SLiM; we check for that case and patch up the breaks vector to reflect what we want to happen in that situation. Finally, we call `addRecombinant()` to generate the offspring bacterium including the horizontal gene transfer. We pass it to the parent genomes – that of the reproducing bacterium, and that of the horizontal gene transfer source – with the breakpoint vector that describes when SLiM should switch between those strands as it produces the offspring genome by recombination. As before, we pass `NULL` for the next three parameters to indicate that the second offspring genome should be a null genome (conceptually, nonexistent). (A diploid model that wanted to generate its own recombination breakpoints might use those parameters, for example.)

Without horizontal gene transfer, this would be a model of clonal competition: one lineage would end up "winning" and the other would go extinct, although it might take a long time for that outcome to be reached since it would depend on drift. This behavior can be seen by setting the defined constant `H` to `0.0`. With horizontal gene transfer, however, the bacteria will often stumble upon a lineage (or perhaps more than one lineage) that combines both mutations in the same genome, providing them with an advantage similar to that provided by recombination in sexual reproduction. Once such a lineage arises it will almost always win, and we will get output like this from the model:

```
197: 2 fixed.
```

Both beneficial mutations fixed in generation 197, thanks to horizontal gene transfer.

The details of the breakpoint generation here might need to be modified in a more realistic model. Here we draw the start and end positions of the transfer region independently, but perhaps it would be better to draw the start location randomly and then draw a transfer length from a geometric distribution or some other distribution. This would constrain the horizontal gene transfer to generally be a small minority of the genome, as is typical in the transfer of a plasmid or a transposon. The location and length of the transfer could also be constrained by some sort of genetic structure to explicitly model the transfer of a plasmid that spans a given range of the genome, of course. The `reproduction()` callback could also base the choice of whether or not horizontal gene transfer occurs upon the contents of the two genomes in question, not just upon a random probability; one could model a selfish gene in the transfer donor that makes horizontal gene transfer more likely to occur, for example. Since all of the logic governing the horizontal gene transfer is in the model's script, it can include whatever biological realism is of interest.

Note that prior to the addition of `addRecombinant()` in SLiM, it would have been possible to model horizontal gene transfer by actually getting all of the mutations from the transfer region out of the source's genome, and then adding them into the target's genome with `addMutations()` (removing any existing mutations from the target region first). This would work fine except that it obscures what is actually going on in terms of genealogy and inheritance. If tree-sequence recording were used with such a model, the transferred region would not be recorded as originating in the source genome; instead, the mutations would just magically appear in the target genome, with no genealogical relationship between source and target recorded in the tree sequence. The method presented here, using `addRecombinant()`, is therefore preferable. (If one needed to model even more complex patterns of inheritance – offspring genomes that consist of a mosaic of genetic material from more than two parental genomes, for example – using the `addMutations()` technique might still be necessary, however, since `addRecombinant()` is designed to record at most two parental genomes for each offspring genome. Tree-sequence recording would not work well in such a model, however.)

To make a relatively realistic model of bacterial evolution with SLiM, this sort of realistic inheritance and horizontal gene transfer is one important ingredient. The other important ingredient is the ability to model a sufficiently large population size, which is often made possible by the performance benefits provided by tree-sequence recording as we will see in chapter 17.

16.15 Implementing a Wright–Fisher model with a nonWF model

In this chapter, we have focused on aspects of nonWF models that go beyond the Wright–Fisher model, such as overlapping generations, age structure, and individual-level control over reproduction and migration. Sometimes, however, it can be useful to implement a Wright–Fisher model as a nonWF model in SLiM – or at least some aspects of a Wright–Fisher model. You might want to have discrete, non-overlapping generations, for example; or you might want panmictic offspring generation as in the Wright–Fisher model, with each offspring being generated from an independent, randomly drawn pair of parents. Implementing such a model using the nonWF model type might still be desirable, because you might also want some non-Wright–Fisher dynamics in your model that would be difficult to implement in a WF model, or you might want to take advantage of certain features of SLiM that are only available in nonWF models. In this section, we will look at two nonWF models that incorporate aspects of the Wright–Fisher model. Both models will include deleterious mutations in addition to neutral mutations, to show how each model treats fitness.

The first recipe here is quite simple, so let's look at it in full:

```
initialize() {
    initializeSLiMModelType("nonWF");
    initializeMutationType("m1", 0.5, "f", 0.0);
    m1.convertToSubstitution = T;
    initializeMutationType("m2", 0.0, "f", -0.5);
    initializeGenomicElementType("g1", c(m1, m2), c(1.0, 0.05));
    initializeGenomicElement(g1, 0, 99999);
    initializeMutationRate(1e-7);
    initializeRecombinationRate(1e-8);
}
reproduction() {
    K = sim.getValue("K");

    // parents are chosen randomly, irrespective of fitness
    parents1 = p1.sampleIndividuals(K, replace=T);
    parents2 = p1.sampleIndividuals(K, replace=T);

    for (i in seqLen(K))
      p1.addCrossed(parents1[i], parents2[i]);

    self.active = 0;
}
1 early() {
    sim.setValue("K", 500);
    sim.addSubpop("p1", sim.getValue("K"));
}
early()
{
    // parents die; offspring survive proportional to fitness
    inds = sim.subpopulations.individuals;
    inds[inds.age > 0].fitnessScaling = 0.0;
}
10000 late() { sim.outputFixedMutations(); }
```