Forge lab 2: Memory Management

Modeling Memory Cells

In this lab, we'll use Forge to model computer memory as a collection of `Memory` cells:

```
sig Memory {}
```

`Memory` may either be a `HeapCell`:

```
sig HeapCell extends Memory {}
```

or appear on the `Stack`:

```
one sig Stack extends Memory {}
```

Since `memory` can only be a `HeapCell` or `Stack`, you should add a predicate `abstractMemory { . . . }` that enforces this constraint.

While there may be many regions of allocated heap memory, there is only *one* contiguous range of stack memory (which we model as a single, unique stack). The `Stack` memory is the starting point from which a program may traverse *references* to reach other parts of memory.

Add these declarations to your model.

Modeling References

So far, the model lacks a notion that memory cells may have *references* to other memory cells. For some applications, it might be suitable to simply add a `references` field to `Memory`:

```
sig Memory { references: set Memory }
```

Try visualizing this model now by adding `run {} for 2 Memory` to your model.

Now change `references: set Memory` to `references: set HeapCell` and rerun `show`. What changed? We will keep the perspective that references only point to `HeapCell`s throughout the lab.

Modeling State

In this lab, we will be considering a type of *automatic memory management* system that automatically frees memory (when it is safe to do so!) as references between memory cells change. To reflect that the references of any given memory cell may change over time, we need to move `references` out of `Memory` and into a new signature representing the memory state *at a particular time*:

```
sig State { references : ... }
```

For this problem, we will only consider two states (`StateA` and `StateC`), plus an intermediate state (`StateB`):

```
one sig StateA extends State {}
one sig StateB extends State {}
One sig StateC extends State {}
```

Note that, because `references` is no longer a field of memory cells, it may have a different type than it had before.

Delete the references field from the `Memory` signature. Add the `State`, `StateA`, `StateB` and `StateC` signature to your model. Replace the ellipsis with the appropriate type.

Again, you'll need to enforce that the only `State` atoms that show up in your instances are `StateA`, `StateB`, and `StateC`. Do so with another predicate, `abstractState { . . . }`.

Visualizer Tip: Projecting over State

Run `show` again.

At first, it may appear that this change has mostly had the effect of making the visualization of the model far more tangled. We would like to easily see what has changed between states. To do this, we can *project* over the `State` signature using the *Projection* menu on the left side of the toolbar of the visualizer.

Project over the State signature. Use the arrows or the dropdown to switch the projection between `StateA` and `StateB`. (If the visualization graph does not contain any edges, click **Next** until you are viewing an instance that does have references.)

Properties of Automatic Memory Management

Automatic memory management systems keep track of which heap memory cells are *allocated* and which are not at each state. We can capture this by adding an `allocated` member to the `State` signature:

```
sig State { allocated : ..., references : ..., }
```

Add the allocated field to the `State` signature in your model. Replace the ellipsis with the appropriate type.

Soundness and Completeness

There are at least two properties we would really like a memory-management system to observe. First, they shouldn't deallocate memory that is still being used (*soundness*). Second, they shouldn't fail to eventually deallocate unreachable/unused memory (*completeness*). Recall that the `Stack` memory is the starting point from which a program may traverse

references to reach other parts of memory. So, if memory is still reachable from the program `Stack`, we're still using it. We hope that our memory management system keeps everything we are using allocated (*soundness*), and tosses everything we are no longer using (*completeness*). To check this, we'll define a helper predicate:

```
pred reachFromStack[m : HeapCell, s : State] { ... }
```

Complete the implementation of `reachFromStack`, which should be true only when `m` (a `HeapCell`) can be reached starting at the `Stack` memory in the given `State`.

Hint: Start by getting the `references` for the current state `s`, which should be a binary relation.

Soundness

The `Stack` memory is the starting point from which a program may traverse references to reach other parts of memory. A memory state is *safe* if all memory reachable from the `Stack` is allocated:

```
pred safe[s : State] { ... }
```

A memory management system is *sound* if acting on an initial *safe* memory state implies that the following state will also be safe:

```
soundness: check { safe[StateA] implies safe[StateC] } for 4 Memory, 3 State
```

Add this predicate and check to your model. Complete the implementation of `safe`. Since you have not described exactly how the memory manager will transition between `StateA` and `StateC`, expect that this check will fail.

Completeness

A memory state is *clean* if all allocated memory is reachable from the `Stack`:

```
pred clean[s : State] { ... }
```

A memory management system is *complete* if acting on an initial *clean* memory state implies that the following state will also be totally collected:

```
completeness: check { clean[StateA] => clean[StateC] } for 4 Memory, 3 State
```

Add this predicate and check to your model. Complete the implementation of `clean`. Since you have not described exactly how the memory manager will transition between `StateA` and `StateC`, expect that this check will fail.

Reference Counting

Reference counting is a form of automatic memory management. For each cell of heap memory, a reference counting collector stores the number of *incoming* references to it. When a memory cell's reference count drops to zero, the cell is unallocated. To represent this, you will

write a function ref_count that returns the number of incoming references to a `HeapCell` in a certain `State`:

```
fun ref_count[m: HeapCell, s: State]: Int { ... }
```

Add and complete the function so that `ref_count` correctly computes the number of incoming references for a cell in a given state. Remember that the `#` operator lets you count the number of entries in relations you build. Alternatively, the `no` operator returns true if and only if given an empty relation.

Implementing Reference Counting Collection

Between `StateA` and `StateB`, the **program** may create or destroy references, but the memory manager does nothing and so the set of allocated cells doesn't change:

```
pred A_to_B_AllocatedUnchanged { ... }
```

Add `A_to_B_AllocatedUnchanged` to your model, and complete the implementation such that the set of allocated memory cells do not change between `StateA` and `StateB`.

Between `StateB` and `StateC`, references should not change, but the allocations may change as a result of **garbage collection**. A reference counting collector will enforce that for all memory cells, a cell will be unallocated in `StateC` iff it has a reference-count of 0 in `StateB`.
`pred B_to_C_GarbageCollected { ... }`

Add `B_to_C_GarbageCollected` to your model, and complete the implementation such that the set of references between memory cells do not change between `StateB` and `StateC`, and the set of allocated memory cells changes appropriately.

Verifying Correctness

Check both `soundness` and `completeness`. Is reference-counting collection sound or complete? If not, why? How do the consequences of violating `soundness` compare to the consequences of violating `completeness`?

Soundness

If you implemented reference counting correctly, soundness *should not* be violated. If you are getting counterexamples, try fixing the problem yourself, or ask a TA for help.

Completeness

Unless you wrote some extra constraints, completeness *should* be violated. If you are not getting counterexamples, try fixing the problem yourself, or ask a TA for help.

Optional: Just for fun

What circumstances might cause completeness to fail? You should be able to guess just by looking at the counterexamples Forge finds for you.

Check-off

Call over a TA and execute your soundness and completeness checks.