Final Script -

Saul:

For this project, our group is translating the small step operational semantics from Review 35 into big step operational semantics. As part of that, we need to look at the values, locations, and memories and what that means in terms of big step.

For the first inference rule, we need to look at address value. This inference rule extends the value judgment form E value to specify that an address is a value.

In other words, address value is used to determine if a given address is associated with a value in memory. It basically says if a memory m maps address a to a value v, then a is a value. This ensures that memory lookups result in actual values when they're needed in evaluations.

As an example, we can look at memory M, which might consist of a1 mapped to 5 and a2 mapped to an array of 1, 2, and 3. Our judgment is that a1 is a value because it's mapped to 5.

For the next judgment form, we can look at the array index location. This judgment form determines if accessing the end element of an array at address a is valid and returns the value or reference. It says if memory m maps to address a, then in an array at a certain n location, a of N is a valid location.

With a different example, we can map a1 to the array [10, 20, 30], ensuring that a specific value maps to a valid location in memory. For instance, one of the values could map to 20.

For the last inference rule or memory definition, we can look at what memory means. This rule describes how memory works as a mapping of addresses to values, or in this case, the creation of an empty memory.

This dot represents an empty memory. Memory can either have addresses mapped to values, or we can create new arrays. Alternatively, we can just have an empty memory.

An example of this is creating a memory where a1 maps to an array we just created. Another important point is that we can add new addresses mapped to different things. We can also add or remove items from memory. If needed, we can add a second memory filled with something else, similar to aliasing.

In terms of big step, we need to ensure that expressions involving addresses in memory and their associated values are valid before proceeding.

For example, if we have an address a and a memory m, we need to make sure that this address evaluates to a value v in this memory before performing the big step to that v.

The judgment of value verifies that a in memory maps to v, and this works the same for an array index location. This rule ensures that memory lookups retrieve the correct value or reference, which might then be used in further big step evaluations.

William:

Search Array Index 1 (Small Step Semantics):
"Ensure that e_1 is evaluated to a new expression & is stored in a new memory location. Then, as we small-step down, the value e_1 will continue to do this until it is a value."

Search Array Index 2 (Small Step Semantics):
"Similar to Search Array Index 1, but the only difference is that e_1 is already a value a_1. So now we have to small-step e_2 down to a value with a new memory location each time."

Do Array / ArrayCreate (Small Step Semantics):
"Here we must make sure that a isn't already in the domain of memory, so it is the premise. Then we have the list of v-values which a (memory address) now points to and is returned."

Do ArrayIndex (Small Step Semantics):
"Given that a is in memory & is equal to [$\bar{v}$], and at index n, the value is V□. Then in the conclusion, we are just changing the value of a at index n to be V□ specifically, and it returns V□."

Do Array Index Assignment (Small Step Semantics):
"Ensure v', a value in m(a), exists which will be assigned a new value. Knowing that it is at index n, set a[n] to the new value V. Then small-step and change the memory of a to contain V instead of v'."

DoArray / Create Array (Big Step Semantics):
Difference:
"The premise that tests every expression in v evaluates to a value V_i."

DoArray Index (Big Step):
Difference:
"The code is pretty much the same, but it directly states that a[n] evaluates to value V in new memory m' without any intermediate steps."

DoArray Index Assignment (Big Step):
Difference:
"The expression e is evaluated to V in the premise."

Danny:

Okay, so now we'll be implementing the code behind the semantic rules that you've already seen for the big step parsing of a re creation. To start, we'll have just the basic definitions of these types. As you can see up here, we have our array.

You can see what everything is equivalent to in the comments. So we have our expressions, array indexing, assigning, and a, which is basically just images. Here are Ray, just some array with all these values in it.

Index would be an array expression indexed at some natural number n. Then in the case of our sign, it would be e 1 as some expression e 1, which is an array, equal to an expression e 2.

This could be combined with index to replace a specific index of an array or an entire array. Now the helper functions, mostly. For this example, we're just going to assume appropriate definitions of is value, is L value, do with, and then copy in this first few.

Here we have is value, is L value, which you can see just representing with the Scala not implemented. Just so we don't have to actually go through and implement these. Here we have our do with. I'll be doing the functions that come with dowith.

There we go. Yes, here's our object do with, with do get, do put, do return, and do modify. And we'll go in and get Mem and memalloc copy those into there. And now we are ready to work on our actual big step itself. Sit down.

Here will be defined big step, taking an expression, and it will return to do with type mem and expression.

As per usual, the first thing we do is match on e, and we'll just need our case for an array declaration so it has a list of expressions in it.

First, what we're gonna do is map on the list because we're gonna wanna check that each element in that array is already a value before we actually go and create the array itself.

So we'll pass in call E, |, and then we'll do if it is not. So if e | is not a value, we're going to want to step on that as well, do a big step, get all the way down till it is value on the eye, and replace the eye with this new big step e.

Otherwise, if it's already a value, if it's already, for instance, an image or 5, we'll just leave it as is. Finally, in our last step, we'll do memalloc(e). So here we have just creating the actual memory space for E as you can see, we have this defined up here.

And like I said, not everything is fully defined. We're just assuming correct implementations for is value, is L value, do with, and mem. Hopefully, this helps and you can understand big step array notation better now in Scala. Thank you for your time.