CSCE604 HW6

Guanlun Zhao

Problem 1.

“Break” and “continue” are implemented using the `<control>` cell. When entering a while loop, the following statements `K`, current environment `Env`, loop statement `S`, loop predicate `E` are stored in the `<lstore>` cell.

I also created a `<lstack>` cell to temporarily keep the `<lstore>` of the outer environment (e.g. outer loop for nested loops), when entering a loop, the current `<lstore>` is added to `<lstack>` and a `endloop;` statement after the loop, which when evaluated, would pop the `<lstore>` from `<lstack>` and restore the current `<lstore>`.

However, if we simply rewrite while loops into other language construct that also contains a while loop, then for every iteration we’d be changing the `<lstack>` and adding a `endloop;` statement. To settle this problem, we created a `rwhile` (recursive while) statement, which does exactly the same as the current while statement. Then we change the while statement as follows:

```
rule <k> while (E) S => rwhile (E) S ~> endloop; ... </k>
       <control>
         <lstack> .List => ListItem(LStore) ...</lstack>
         <lstore> LStore </lstore>
         C
       </control>
```

Here we inserted `endloop;` only when dealing with the while loop and subsequent `rwhlie` statement would not incur such changes. This ensures the environment is only restored once for each loop.

When breaking from a loop, the remaining statements are ignores and the following statements `K` is brought back to the `<k>` cell. For “continue”, instead of putting only K onto the active computation, we also add a new loop `rwhile (E) S` so that the loop would be executed one more time.

As for loops behave differently from while loops when “`continue`” happens (index increment is still performed), I also stored the increment step into the `<lstore>` cell, so that if the current loop is a for loop, in additional to adding `rwhile (E) S ~> K` to the `<k>` cell, I also prepend that with the increment operation so that it would look something like `i++; ~> rwhile (E) S ~> K`.

I also suspect that the K framework has a bug regarding the `<control>` cell which makes me unable to use ListItems in the `<lstore>` cells, which I reported at https://piazza.com/class/ixc6cki6i9r6dl?cid=89. This makes it difficult to use <lstack> directly to store the loop variables.

I've tested all the provided test cases and all the output are correct except for the test case `collatz-continue.simple`. This is because <lstore> and <lstack> behaviors at function calls are not handled 100% correct, partly due to the aforementioned bug.

Problem 2.

Problem 2 is relatively straightforward. I simply added type checking when an exception is thrown:

```
rule <k> throw V:Val; ~> _ => if (typeOf(V) =/=K T) {throw V;} else { T X = V; S2 } ~> K
</k>
```

If the exception type does not match, the current exception handling logic is disregarded and the exception is thrown once more for some outer exception handling context to handle.

I checked the type of the thrown exception and compare it with the type of the current `catch` statement. If they does not match, the exception would be thrown to the outer context.

Bonus Problem.

I chose to work on the bonus problem instead of P3.
My approach is to check whether a variable is secret at the print statement by adding a `<secrecy>` cell and remember all the variables declared secret. However, our current print statement is strict, which means variables would have been converted to their actual values before we could query whether it is secret.

To solve that problem we make print non-strict and introduce another rule called "`realPrint`" (named after a Twitter celebrity) which is strict. "`readPrint`" does exactly what "print" used to do.

```
rule <k> print(V, Es); => if (V in keys(Secrecy)) { realprint("DANGEROUS!"); halt; } ~>
realprint(V, Es); ...</k>
       <secrecy> Secrecy </secrecy>
```

Now that "`print`" is no longer strict, we can check the ID it takes and if that ID is declared secret, we could simply "`realPrint`" the error message and abort the program. Otherwise it is simply re-written to "`realPrint`" and whatever needs to be printed would be printed.

The next problem is the propagate secret values along the data-flow path. We observe that when looking up an ID `X` we could check whether it is secret, and when assigning a value to an

ID through `lvalue(Y)`, we could mark `Y` as secret as well. Therefore we introduce another cell `<ongoing>` which stores only a boolean variable indicating whether a secret computation has just happened. We looking up a secret ID we'd set that flag to true and later on at the lvalue rule we'd mark `Y` as secret, and then set the flag back to false.

Doing this we'd be able to handle the data-flow problem and report "DANGEROUS" when attempting to print out a secret value.