**AOD Implementation**

AOD is replaces an arithmetic expression by each one of the operand. Below is the replacement table which is applied in the scope of the program:

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Expression sample | Replaced by the first operand | Replaced by the second operand |
| + | a + b | a | b |
| - | a - b |
| \* | a \* b |
| / | a / b |
| % | a % b |

1. First operand replacement

To replace an expression by the first operand, we implement to remove the first parameter from the stack of java virtual machine which is associated the second operand in the expression. However, for each data type, there is a different way to store them into stack. With from 8 to 32 bit data, we need to store in 1 stack slot, while 64 bit data needs 2 stack slots. Therefore, It would take 1 stack slot for *int* and *float* and this figure is 2 stack slots for *double* and *long.* Thus, we use the operand stack *POP* to remove item from the stack for *int* and *float* operators and the *POP2* for *double* and *long* operators*.* The code snippet below shows the example of implementation for addition operator:

1. // ADD   
   MUTATIONS.put(Opcodes.IADD, new InsnSubstitution(Opcodes.POP, "AOD Mutator: Removed the second operator from an addition formula (int)"));  
   MUTATIONS.put(Opcodes.LADD, new InsnSubstitution(Opcodes.POP2, "AOD Mutator: Removed the second operator from an addition formula (long)"));
2. Second operand replacement

To replace an expression by the second operand, there are 2 scenarios. For the int and float operators, we use operand *SWAP* to swap two items on the stack. Then, the order of operand in the expression is changed and the problem becomes the replace the expression by the first operand. In case of long and double operators, there doesn’t exist any operand to swap first two slots with next two slots in the stack. So that we use *DUP2\_X2* to duplicate the first two slots in the stack and push them after the 4-th slot in the stack. Then we use *POP2* two timesto pop first 4-slots on the top of the stack and the remainder is the second operand of the expression. Below is the code snippet for those above scenarios:

1. // For int and long operators  
      super.visitInsn(Opcodes.SWAP);  
      super.visitInsn(Opcodes.POP);  
   // For long and double operators  
      super.visitInsn(Opcodes.DUP2\_X2);  
      super.visitInsn(Opcodes.POP2);

super.visitInsn(Opcodes.POP2);

Here is our sample of running AOD mutators with the testcase :

1. public  void  testApp()  {
2. App  app  =  new  App();
3. assertEquals(2,  app.sumInt(1,  1));
4. assertEquals(3,  app.modLong(8,  5));
5. assertEquals(0.0,  app.mulDouble(1.0,  0.0));
6. }

