SSLL Testing Strategy

Paul Nickerson

Fuzz-testing

For the testing requirements of parts 1 and 2, I wrote a list fuzzer that applies pseudo-random operations (starting from a seed value so fuzz runs are reproducable) to each of the four list types, then check to ensure consistency between the lists. An operation is defined as invoking some public list member with a random input. Ideally, the various fuzzing operations should theoretically combine to maximize code coverage. Consistent behavior is defined using the following criteria: after an operation is applied to each list,

- Each list must return an equivalent integer value (analogous to an executable's return code) from the operation. Typically this is the return value of size() called after the method is invoked, but it could also be, for example, a boolean-to-integer cast of the return value from calling contains() or is_empty()
- Each list must report equivalent list contents as determined by the print() function
- If an exception is thrown, the what() method called on each exception must return the same value.
- If the operation involves returning an item, such as calling remove() or item_at(), each operation must return the same item as determined by that item's operator==() member

Operations are generally only a few lines long:

```
size_t i = rand_int(l.size());
l[i] = rv();
ret_item = l.item_at(i);
return l.size();
```

In practice this approach is extremely effective; operations are combined in ways that may be unintuitive for a person trying to come up with testing strategies. It is important, however, to target potentially problematic code paths, such as the growing/shrinking code used by some of the lists. Whenever I wanted to be sure to test these code paths, I would grow and shrink the list acutely:

```
for (int i = 0; i < 1000; ++i) {
    l.push_front(rv());
    l.push_back(rv());
    int sz = l.size();
    int rand_slot = rand_int(sz);
    l.insert(rv(), rand_slot);
}
for (int i = 0; i < 800; ++i) {
    ret_item = ret_item + l.pop_front();
    ret_item = ret_item + l.pop_back();
    int sz = l.size();
    int rand_slot = rand_int(sz);
    ret_item = ret_item + l.remove(rand_slot);
}
```

Fuzzer architecture

Because usage of polymorphism in the list classes was explicitly prohibited, I wrote a class template, ListFuzzer, which takes a list type as its template parameter. Fuzzer operations are defined as a list of lambda functions and are stored in any of the implemented lists. Since, for example, ListFuzzer<SSLL<...» and ListFuzzer<CDAL<...» are different types and cannot be directly compared, the operation invoker returns an OpResult struct instance, which describes the operation, the list name in std::string format, and the results of performing the operation on that list. As such, there are four OpResults generated during each fuzzing testcase - one for each list type. These are passed via initializer list to the result validation function, which verifies consistency. After each round, the source code of the operation is read from the source code file (determined with the ___LINE___ and ___FILE___ preprocessor macros) and displayed on the console along with the results from running the operation.

The fuzzer increments a counter after each fuzzing round. Whenever inconsistent results are returned, the fuzzer throws an exception. While debugging, this behavior is extremely handy; since all fuzzing behavior stems from a single seed specified at the beginning, the user can set a conditional breakpoint to break when the operation counter equals the value it was when inconsistent state was detected, but before the operation that caused the inconsistent state to occur actually gets called. The user can then step into the suspect code and find the culprit.

Because the fuzzer expects strict consistency of list behavior, even among the text of exceptions that are thrown, using the fuzzer as a supplement to development forced me to use similar patterns between the lists. Several methods, such as the subscript operator, the print function, and the copy constructor, have converged on versions which use exactly the same code in each of the lists. This reduces the number of possible things that can go wrong.

Results

Using the fuzzer regularly during list development uncovered several subtle bugs which likely would not have been found during simple tests. According to Murphy's Law the bugs would therefore have been encountered only in production when the misbehavior of the list would lead to nuclear missiles being launched at orphanages. At first, bugs would be found right away, within the first hundred-or-so operations. As I fixed bugs and refactored list code, bugs started appearing much less frequently. At their current state, the lists appear to be rock-solid and display consistent behavior even after hundreds of millions of fuzzer operations.