Pushing Lit's Boundaries to Test Libc++

Louis Dionne

Outline

1. Introduction to Lit
2. The libc++ testing format
3. Usage examples
4. Improvements to Lit

Lit: LLVM Integrated Tester

- Test runner used for most tests across LLVM
- Standalone tool for use outside LLVM
- Simple Python codebase in `llvm/utils/lit`

Basic usage

```
$ lit [options...] test-suites...
```

Wonder how `check-all` works? Basically:

```
$ lit -sv libcxx/test clang/test ...
```

⭐ Omitting `lit.site.cfg` generation by CMake for simplicity

How it works

```
$ lit -sv libcxx/test
```

1. Discover test suites (`lit.cfg` or `lit.site.cfg`)
2. Load the test suite configuration (by executing the `.cfg`)
3. Discover tests by traversing `libcxx/test`
4. Run the tests

"Run" the tests?

In the `.cfg` file:

```
config.name = 'some-name'
config.test_source_root = '<where-to-discover-tests>'
config.test_format = lit.formats.ShTest()
config.test_exec_root = '<where-to-execute-tests>'
config.substitutions = [...]
```

`ShTest` is a Lit **Test Format**

What's a Test Format?

```
class MyTestFormat(lit.formats.TestFormat):
  def execute(self, test, litConfig):
    # Do some stuff...
    return lit.Test.Result(lit.Test.PASS)
```

Some other result codes:

PASS FAIL UNSUPPORTED

XFAIL XPASS TIMEOUT

The ShTest format

```
// UNSUPPORTED: some-system

// RUN: clang++ %s -o a.out
// RUN: ./a.out

int main() { }
```

The ShTest format

```
class ShTest(lit.formats.TestFormat):
  def execute(self, test, litConfig):
    # 1. Parse REQUIRES, UNSUPPORTED, RUN, etc.
    # 2. Substitute %clang_cc1, %s, etc. in RUN
    # 3. Execute RUN lines as a shell script
    return lit.TestRunner.executeShTest(
                              test, litConfig)
```

About substitutions

Custom substitutions in the `.cfg` file:

```
config.substitutions = [
  ('%clang', '/path/to/clang'),
  ('%clang_cc1', '/path/to/clang -cc1'),
  ...
]
```

About substitutions

Also several builtin substitutions:

`%s`	path to the current file
`%S`	path to the current directory
`%t`	unique temporary file name

⭐ See https://llvm.org/docs/CommandGuide/lit.html#substitutions for full list

A more realistic example

```
// RUN: %clang_cc1 -triple x86_64-unknown-unknown   \
// RUN:          -fexceptions -fcxx-exceptions -O0  \
// RUN:          -fno-elide-constructors -emit-llvm \
// RUN:          %s -o - | FileCheck %s

int main() {
  try {
    Container c1;
    // CHECK: ...
    // CHECK-NOT: ...
    Container c2(c1);

    return 2;
  } catch (...) {
    return 1;
  }
  return 0;
}
```

The old libc++ format

```
class LibcxxTestFormat(object):
    def execute(self, test, lit_config):
        while True:
            try:
                return self._execute(test, lit_config)
            except OSError as oe:
                if oe.errno != errno.ETXTBSY:
                    raise
                time.sleep(0.1)

    def _execute(self, test, lit_config):
        name = test.path_in_suite[-1]
        name_root, name_ext = os.path.splitext(name)
        is_libcxx_test = test.path_in_suite[0] == 'libcxx'
        is_sh_test = name_root.endswith('.sh')
        is_pass_test = name.endswith('.pass.cpp') or name.endswith('.pass.mm')
        is_fail_test = name.endswith('.fail.cpp')
        is_objcxx_test = name.endswith('.mm')
        assert is_sh_test or name_ext == '.cpp' or name_ext == '.mm', \
            'non-cpp file must be sh test'

        if test.config.unsupported:
            return (lit.Test.UNSUPPORTED,
                    "A lit.local.cfg marked this unsupported")

        if is_objcxx_test and not \
           'objective-c++' in test.config.available_features:
            return (lit.Test.UNSUPPORTED, "Objective-C++ is not supported")

        setattr(test, 'file_dependencies', [])
        parsers = self._make_custom_parsers(test)
        script = lit.TestRunner.parseIntegratedTestScript(
            test, additional_parsers=parsers, require_script=is_sh_test)

        # Check if a result for the test was returned. If so return that
        # result.
        if isinstance(script, lit.Test.Result):
            return script
        if lit_config.noExecute:
            return lit.Test.Result(lit.Test.PASS)

        # Check that we don't have run lines on tests that don't support them.
        if not is_sh_test and len(script) != 0:
            lit_config.fatal('Unsupported RUN line found in test %s' % name)

        tmpDir, tmpBase = lit.TestRunner.getTempPaths(test)
        substitutions = lit.TestRunner.getDefaultSubstitutions(
            test, tmpDir, tmpBase, normalize_slashes=True)

        # Apply substitutions in FILE_DEPENDENCIES markup
        data_files = lit.TestRunner.applySubstitutions(test.file_dependencies,
substitutions,

recursion_limit=test.config.recursiveExpansionLimit)
        local_cwd = os.path.dirname(test.getSourcePath())
        data_files = [f if os.path.isabs(f) else os.path.join(local_cwd, f) for
f in data_files]
        substitutions.append(('%{file_dependencies}', ' '.join(data_files)))

        # Add other convenience substitutions
        substitutions.append(('%{build}', '%{cxx} -o %t.exe %s %{flags} %
{compile_flags} %{link_flags}'))
        substitutions.append(('%{run}', '%{exec} %t.exe'))

        script = lit.TestRunner.applySubstitutions(script, substitutions,

recursion_limit=test.config.recursiveExpansionLimit)

        test_cxx = copy.deepcopy(self.cxx)
        if is_fail_test:
            test_cxx.useCCache(False)
            test_cxx.useWarnings(False)
        if '-fmodules' in test.config.available_features:
            test_cxx.addWarningFlagIfSupported('-Wno-macro-redefined')
            # FIXME: libc++ debug tests #define _LIBCPP_ASSERT to override it
            # If we see this we need to build the test against uniquely built
            # modules.
            if is_libcxx_test:
                with open(test.getSourcePath(), 'rb') as f:
                    contents = f.read()
                if b'#define _LIBCPP_ASSERT' in contents:
                    test_cxx.useModules(False)

        # Handle ADDITIONAL_COMPILE_FLAGS keywords by adding those compilation
        # flags, but first perform substitutions in those flags.
        extra_compile_flags = self._get_parser('ADDITIONAL_COMPILE_FLAGS:',
parsers).getValue()
        extra_compile_flags =
lit.TestRunner.applySubstitutions(extra_compile_flags, substitutions)
        test_cxx.compile_flags.extend(extra_compile_flags)

        if is_objcxx_test:
            test_cxx.source_lang = 'objective-c++'
            test_cxx.link_flags += ['-framework', 'Foundation']

        # Dispatch the test based on its suffix.
        if is_sh_test:
            if not isinstance(self.executor, LocalExecutor) and not
isinstance(self.executor, SSHExecutor):
                # We can't run ShTest tests with other executors than
                # LocalExecutor and SSHExecutor yet.
                # For now, bail on trying to run them
                return lit.Test.UNSUPPORTED, 'ShTest format not yet supported'
            test.config.environment =
self.executor.merge_environments(os.environ, self.exec_env)
            return lit.TestRunner._runShTest(test, lit_config,
                                             True, script,
                                             tmpBase)
        elif is_fail_test:
            return self._evaluate_fail_test(test, test_cxx, parsers)
        elif is_pass_test:
            return self._evaluate_pass_test(test, tmpBase, lit_config,
                                            test_cxx, parsers, data_files)
        else:
            # No other test type is supported
            assert False

    def _clean(self, exec_path):  # pylint: disable=no-self-use
        libcxx.util.cleanFile(exec_path)

    def _evaluate_pass_test(self, test, tmpBase, lit_config,
                            test_cxx, parsers, data_files):
        execDir = os.path.dirname(test.getExecPath())
        source_path = test.getSourcePath()
        exec_path = tmpBase + '.exe'
        object_path = tmpBase + '.o'
        # Create the output directory if it does not already exist.
        libcxx.util.mkdir_p(os.path.dirname(tmpBase))
        try:
            # Compile the test
            cmd, out, err, rc = test_cxx.compileLinkTwoSteps(
                source_path, out=exec_path, object_file=object_path,
                cwd=execDir)
            compile_cmd = cmd
            if rc != 0:
                report = libcxx.util.makeReport(cmd, out, err, rc)
                report += "Compilation failed unexpectedly!"
                return lit.Test.Result(lit.Test.FAIL, report)
            # Run the test
            env = None
            if self.exec_env:
                env = self.exec_env

            max_retry = test.allowed_retries + 1
            for retry_count in range(max_retry):
                # Create a temporary directory just for that test and run the
                # test in that directory
                try:
                    execDirTmp = tempfile.mkdtemp(dir=execDir)
                    cmd, out, err, rc = self.executor.run(exec_path,
[exec_path],
                                                          execDirTmp,
data_files,
                                                          env)
                finally:
                    shutil.rmtree(execDirTmp)
                report = "Compiled With: '%s'\n" % ' '.join(compile_cmd)
                report += libcxx.util.makeReport(cmd, out, err, rc)
                if rc == 0:
                    res = lit.Test.PASS if retry_count == 0 else
lit.Test.FLAKYPASS
                    return lit.Test.Result(res, report)
                elif rc != 0 and retry_count + 1 == max_retry:
                    report += "Compiled test failed unexpectedly!"
                    return lit.Test.Result(lit.Test.FAIL, report)

            assert False # Unreachable
        finally:
            # Note that cleanup of exec_file happens in `_clean()`. If you
            # override this, cleanup is your reponsibility.
            libcxx.util.cleanFile(object_path)
            self._clean(exec_path)

    def _evaluate_fail_test(self, test, test_cxx, parsers):
        source_path = test.getSourcePath()
        # FIXME: lift this detection into LLVM/LIT.
        with open(source_path, 'rb') as f:
            contents = f.read()
        verify_tags = [b'expected-note', b'expected-remark',
                       b'expected-warning', b'expected-error',
                       b'expected-no-diagnostics']
        use_verify = self.use_verify_for_fail and \
                     any([tag in contents for tag in verify_tags])
        test_cxx.flags += ['-fsyntax-only']
        if use_verify:
            test_cxx.useVerify()
        cmd, out, err, rc = test_cxx.compile(source_path, out=os.devnull)
        check_rc = lambda rc: rc == 0 if use_verify else rc != 0
        report = libcxx.util.makeReport(cmd, out, err, rc)
        if check_rc(rc):
            return lit.Test.Result(lit.Test.PASS, report)
        else:
            report += ('Expected compilation to fail!\n' if not use_verify else
                       'Expected compilation using verify to pass!\n')
            return lit.Test.Result(lit.Test.FAIL, report)
```

The new libc++ test format

```
class CxxStandardLibraryTest(lit.formats.TestFormat):
  def execute(self, test, litConfig):
    filename = test.path_in_suite[-1]
    if filename.endswith('.compile.fail.cpp'):
      steps = ["! %{cxx} %s %{flags} %{compile_flags} -fsyntax-only"]
    elif filename.endswith('.verify.cpp'):
      steps = ["%{cxx} %s %{flags} %{compile_flags} -fsyntax-only -Xclang -verify"]
    elif filename.endswith('.pass.cpp'):
      steps = [
        "%{cxx} %s %{flags} %{compile_flags} %{link_flags} -o %t.exe",
        "%{exec} %t.exe"
      ]
    elif ...:
      ...
    else:
      return lit.Test.Result(lit.Test.UNRESOLVED, "Unknown test suffix")

    return lit.TestRunner.executeShTest(test, litConfig, preamble_commands=steps)
```

⭐ This is of course a simplification meant to make the new format look better

Supports new kinds of tests

	Compile	Link	Run
*.pass.cpp	✅	✅	✅
*.run.fail.cpp	✅	✅	❌
*.link.pass.cpp	✅	✅	n/a
*.link.fail.cpp	✅	❌	n/a
*.compile.pass.cpp	✅	n/a	n/a
*.compile.fail.cpp	❌	n/a	n/a
*.sh.cpp	n/a	n/a	n/a
*.verify.cpp	✅	n/a	n/a

Configuration

The basic configuration, plus:

```
config.test_format = libcxx.test.format.CxxStandardLibraryTest()
config.substitutions = [
  ('%{cxx}', 'clang++'),
  ('%{flags}', ''),
  ('%{compile_flags}', '-nostdinc++ -isystem {}/include/c++/v1'
                                                .format(LIBCXX_ROOT)),
  ('%{link_flags}', '-nostdlib++ -L {0}/lib -lc++ -Wl,-rpath,{0}/lib'
                                                .format(INSTALL_ROOT)),
  ('%{exec}', '{}/utils/run.py --execdir %T -- '.format(LIBCXX_ROOT))
]
```

More substitutions

The format itself defines:

```
config.substitutions += [
  ('%{build}', '%{cxx} %s %{flags} %{compile_flags} \
                          %{link_flags} -o %t.exe'),
  ('%{run}', '%{exec} %t.exe')
]
```

Flexibility #1

Running the tests against static libraries:

```
config.test_format = libcxx.test.format.CxxStandardLibraryTest()
config.substitutions = [
  ('%{cxx}', 'clang++'),
  ('%{flags}', ''),
  ('%{compile_flags}', '-nostdinc++ -isystem {}/include/c++/v1'
                                         .format(LIBCXX_ROOT)),
  ('%{link_flags}', '-nostdlib++ {0}/lib/libc++.a \
                                 {0}/lib/libc++abi.a'
                                         .format(INSTALL_ROOT)),
  ('%{exec}', '{}/utils/run.py --execdir %T -- '
                                         .format(LIBCXX_ROOT))
]
```

Flexibility #2

Running the tests on a remote device:

```
config.test_format = libcxx.test.format.CxxStandardLibraryTest()
config.substitutions = [
  ('%{cxx}', 'clang++'),
  ('%{flags}', '--target=armv7-linux-gnueabihf'),
  ('%{compile_flags}', '-nostdinc++ -isystem {}/include/c++/v1'
                                         .format(LIBCXX_ROOT)),
  ('%{link_flags}', '-nostdlib++ {0}/lib/libc++.a \
                                 {0}/lib/libc++abi.a'
                                         .format(INSTALL_ROOT)),
  ('%{exec}', '{}/utils/ssh.py --host="foo@xyz.lab.llvm.org" \
                               --execdir %T -- '
                                         .format(LIBCXX_ROOT))
]
```

Flexibility #3

Running the tests against another standard library:

```
config.test_format = libcxx.test.format.CxxStandardLibraryTest()
config.substitutions = [
  ('%{cxx}', '{}/bin/g++'.format(GCC_ROOT)),
  ('%{flags}', ''),
  ('%{compile_flags}', ''),
  ('%{link_flags}', '-Wl,-rpath,{}/lib64'.format(GCC_ROOT)),
  ('%{exec}', '{}/utils/run.py --execdir %T -- '
                                         .format(LIBCXX_ROOT))
]
```

Interesting ideas

- Reuse the `ShTest` format for most logic
- Substitutions are *inputs* of the test format
- Build substitutions on top of others

Problems solved

1. Inscrutable config files
2. Constant flow of funky use cases
3. Painful stringing of options through CMake

Improvements to Lit

Support for flaky tests

(please don't use)

Lit will allow the specified number of retries

```
// ALLOW_RETRIES: 2

int main(int, char**) {
  // Do something slightly timing-dependent
  std::thread t1 = []() { ... };
  std::thread t2 = []() { ... };
  t1.join();
  t2.join();
  assert(...);
}
```

Recursive substitutions

Allows expanding substitutions inside substitutions:

```
// RUN: %{build}

  => // RUN: %{cxx} %s %{flags} -o %t.exe

  => // RUN: clang++ foobar.cpp -std=c++17 -Wall

                                -o /tmp/foo.exe
```

Cleaning up test suites

Allows figuring out which Lit features are used:

```
$ lit --show-used-features libcxx/test
-faligned-allocation -fmodules -fno-rtti -fsized-deallocation LIBCXX-
WINDOWS-FIXME apple-clang apple-clang-10 apple-clang-10.0 apple-
clang-10.0.0 apple-clang-11 apple-clang-11.0.0 apple-clang-12 apple-
clang-9 apple-clang-9.0 apple-clang-9.1 asan availability=macosx10.10
availability=macosx10.11 availability=macosx10.12
availability=macosx10.13 availability=macosx10.14
availability=macosx10.15 availability=macosx10.9 c++03 c++11 c++14 c+
+17 c++2a c++filesystem-disabled clang clang-10 clang-4 clang-4.0
clang-5 clang-5.0 clang-6 clang-6.0 clang-7 clang-7.0 clang-8
clang-8.0 clang-9 darwin diagnose-if-support fcoroutines-ts fdelayed-
template-parsing gcc gcc-5 gcc-5.1 gcc-5.2 gcc-6 gcc-7
gcc-8 gcc-9 has-fblocks has-fobjc-arc libc++ libcpp-has-no-global-
filesystem-namespace [...]
```

ShTest preamble commands

Allows running commands before a ShTest

```
lit.TestRunner.executeShTest(test, litConfig,
                             preamble_commands=[
  'echo This is run before the test',
  'echo I can setup something here'
])
```

Thank you