Testing Guide

Quickstart commands ¶

These commands are explained below in more detail.

Run the main MLIR test suite: ¶

```
cmake --build . --target check-mlir
```

Run integration tests (requires `-DMLIR_INCLUDE_INTEGRATION_TESTS=ON`): ¶

```
cmake --build . --target check-mlir-integration
```

Run C++ unit tests: ¶

```
cmake --build . --target check-mlir-unit
```

Run `lit` tests in a specific directory ¶

```
# from build/
# -v shows output of failed tests
bin/llvm-lit -v tools/mlir/test/Dialect/Arithmetic
```

Run a specific `lit` test file ¶

```
# from build/
# -v shows output of failed tests
bin/llvm-lit -v
tools/mlir/test/Dialect/Polynomial/ops.mlir
```

Test categories ¶

`lit` and `FileCheck` tests ¶

`FileCheck` is a tool that “reads two files (one from standard input, and one specified on the command line) and uses one to verify the other.” One file contains a set of `CHECK` tags that specify strings and patterns expected to appear in the other file. MLIR utilizes `FileCheck`, in combination with `lit`, to verify different aspects of the IR—such as the output of a transformation pass.

The source files of `lit` / `FileCheck` tests are organized within the `mlir` source tree under `mlir/test/`. Within this directory, tests are organized roughly mirroring `mlir/include/mlir/`, including subdirectories for `Dialect/`, `Transforms/`, `Conversion/`, etc.

Example ¶

An example `FileCheck` test is shown below:

```
// RUN: mlir-opt %s -cse | FileCheck %s

// CHECK-LABEL: func.func @simple_constant
func.func @simple_constant() -> (i32, i32) {
  // CHECK-NEXT: %[[RESULT:.*]] = arith.constant 1
  // CHECK-NEXT: return %[[RESULT]], %[[RESULT]]

  %0 = arith.constant 1 : i32
  %1 = arith.constant 1 : i32
  return %0, %1 : i32, i32
}
```

A comment with `RUN` represents a `lit` directive specifying a command line invocation to run, with special substitutions like `%s` for the current file. A comment with `CHECK` represents a `FileCheck` directive to assert a string or pattern appears in the output.

The above test asserts that, after running Common Subexpression Elimination (`-cse`), only one constant remains in the IR, and the sole SSA value is returned twice from th function.

Build system details ¶

While developing in MLIR, it is common to run parts of the test suite many times. While invoking the `check-mlir` target is a

nice shortcut, the interaction with the build system can be confusing. Since most people don't understand how all of this is put together in the build system, this section attempts to demystify how to operate the testing tools independently.

Invoking the `check-mlir` target is roughly equivalent to running (from the build directory):

```
./bin/llvm-lit tools/mlir/test
```

See the Lit Documentation for a description of all options.

Subsets of the testing tree can be invoked by passing a more specific path instead of `tools/mlir/test` above. Example:

```
./bin/llvm-lit tools/mlir/test/Dialect/Arithmetic

# Note that it is possible to test at the file granularity, but since these
# files do not actually exist in the build directory, you need to know the
# name.
./bin/llvm-lit tools/mlir/test/Dialect/Arithmetic/ops.mlir
```

Lit has a number of options that control test execution. Here are some of the most useful for development purposes:

- `--filter=REGEXP` : Only runs tests whose name matches the REGEXP. Can also be specified via the `LIT_FILTER` environment variable.
- `--filter-out=REGEXP` : Filters out tests whose name matches the REGEXP. Can also be specified via the `LIT_FILTER_OUT` environment variable.

- `-a` : Shows all information (useful while iterating on a small set of tests).
- `--time-tests` : Prints timing statistics about slow tests and overall histograms.

Any Lit options can be set in the `LIT_OPTS` environment variable. This is especially useful when using the build system target `check-mlir`.

Examples:

```
# Only run tests that have "python" in the name
and print all invocations.
LIT_OPTS="--filter=python -a" cmake --build . --
target check-mlir

# Only run the array_attributes python test, using
the LIT_FILTER mechanism.
LIT_FILTER="python/ir/array_attributes" cmake --
build . --target check-mlir

# Run everything except for example and
integration tests (which are both
# somewhat slow).
LIT_FILTER_OUT="Examples|Integrations" cmake --
build . --target check-mlir
```

Note that the above use the generic cmake command for invoking the `check-mlir` target, but you can typically use the generator directly to be more concise (i.e. if configured for `ninja`, then `ninja check-mlir` can replace the `cmake --build . --target check-mlir` command). We use generic `cmake` commands in documentation for consistency, but being concise is often better for interactive workflows.

Diagnostic tests ¶

MLIR provides rich source location tracking that can be used to emit errors, warnings, etc. from anywhere throughout the codebase, which are jointly called *diagnostics*. Diagnostic tests assert that specific diagnostic messages are emitted for a given input program. These tests are useful in that they allow checking specific invariants of the IR without transforming or changing anything.

Some examples of tests in this category are:

- Verifying invariants of operations
- Checking the expected results of an analysis
- Detecting malformed IR

Diagnostic verification tests are written utilizing the source manager verifier handler, which is enabled via the `verify-diagnostics` flag in `mlir-opt`.

An example .mlir test running under `mlir-opt` is shown below:

```
// RUN: mlir-opt %s -split-input-file -verify-diagnostics

// Expect an error on the same line.
func.func @bad_branch() {
  cf.br ^missing  // expected-error {{reference to an undefined block}}
}

// -----

// Expect an error on an adjacent line.
func.func @foo(%a : f32) {
```

```
  // expected-error@+1 {{invalid predicate
attribute specification: "foo"}}
  %result = arith.cmpf "foo", %a, %a : f32
  return
}
```

Integration tests ¶

Integration tests are `FileCheck` tests that verify functional correctness of MLIR code by running it, usually by means of JIT compilation using `mlir-cpu-runner` and runtime support libraries.

Integration tests don't run by default. To enable them, set the `-DMLIR_INCLUDE_INTEGRATION_TESTS=ON` flag during `cmake` configuration as described in Getting Started.

```
cmake -G Ninja ../llvm \
   ... \
   -DMLIR_INCLUDE_INTEGRATION_TESTS=ON \
   ...
```

Now the integration tests run as part of regular testing.

```
cmake --build . --target check-mlir
```

To run only the integration tests, run the `check-mlir-integration` target.

```
cmake --build . --target check-mlir-integration
```

The source files of the integration tests are organized within the `mlir` source tree by dialect (for example, `test/Integration/Dialect/Vector`).

Hardware emulators ¶

The integration tests include some tests for targets that are not widely available yet, such as specific AVX512 features (like `vp2intersect`) and the Intel AMX instructions. These tests require an emulator to run correctly (lacking real hardware, of course). To enable these specific tests, first download and install the Intel Emulator. Then, include the following additional configuration flags in the initial set up (X86Vector and AMX can be individually enabled or disabled), where `<path to emulator>` denotes the path to the installed emulator binary.

```
cmake -G Ninja ../llvm \
   ... \
   -DMLIR_INCLUDE_INTEGRATION_TESTS=ON \
   -DMLIR_RUN_X86VECTOR_TESTS=ON \
   -DMLIR_RUN_AMX_TESTS=ON \
   -DINTEL_SDE_EXECUTABLE=<path to emulator> \
   ...
```

After this one-time set up, the tests run as shown earlier, but will now include the indicated emulated tests as well.

Unit tests ¶

Unit tests are written using the googletest framework and are located in the `mlir/unittests/` directory.

Contributor guidelines ¶

In general, all commits to the MLIR repository should include an accompanying test of some form. Commits that include no functional changes, such as API changes like symbol renaming, should be tagged with NFC (No Functional Changes). This

signals to the reviewer why the change doesn't/shouldn't include a test.

`lit` tests with `FileCheck` are the preferred method of testing in MLIR for non-erroneous output verification.

Diagnostic tests are the preferred method of asserting error messages are output correctly. Every user-facing error message (e.g., `op.emitError()`) should be accompanied by a corresponding diagnostic test.

When you cannot use the above, such as for testing a non-user-facing API like a data structure, then you should write unit tests. This is preferred because the C++ APIs are not stable and subject to frequent refactoring. Using `lit` and `FileCheck` allows maintainers to improve the MLIR internals more easily.

FileCheck best practices ¶

FileCheck is an extremely useful utility, it allows for easily matching various parts of the output. This ease of use means that it becomes easy to write brittle tests that are essentially `diff` tests. FileCheck tests should be as self-contained as possible and focus on testing the minimal set of functionalities needed. Let's see an example:

```
// RUN: mlir-opt %s -cse | FileCheck %s

// CHECK-LABEL: func.func @simple_constant() ->
(i32, i32)
func.func @simple_constant() -> (i32, i32) {
  // CHECK-NEXT: %result = arith.constant 1 : i32
  // CHECK-NEXT: return %result, %result : i32,
i32
  // CHECK-NEXT: }
```

```
  %0 = arith.constant 1 : i32
  %1 = arith.constant 1 : i32
  return %0, %1 : i32, i32
}
```

The above example is another way to write the original example shown in the main FileCheck tests section. There are a few problems with this test; below is a breakdown of the no-nos of this test to specifically highlight best practices.

- Tests should be self-contained.

This means that tests should not test lines or sections outside of what is intended. In the above example, we see lines such as `CHECK-NEXT: }`. This line in particular is testing pieces of the Parser/Printer of FuncOp, which is outside of the realm of concern for the CSE pass. This line should be removed.

- Tests should be minimal, and only check what is absolutely necessary.

This means that anything in the output that is not core to the functionality that you are testing should *not* be present in a CHECK line. This is a separate bullet just to highlight the importance of it, especially when checking against IR output.

If we naively remove the unrelated `CHECK` lines in our source file, we may end up with:

```
// CHECK-LABEL: func.func @simple_constant
func.func @simple_constant() -> (i32, i32) {
  // CHECK-NEXT: %result = arith.constant 1 : i32
  // CHECK-NEXT: return %result, %result : i32,
i32

  %0 = arith.constant 1 : i32
```

```
  %1 = arith.constant 1 : i32
  return %0, %1 : i32, i32
}
```

It may seem like this is a minimal test case, but it still checks several aspects of the output that are unrelated to the CSE transformation. Namely the result types of the `arith.constant` and `return` operations, as well the actual SSA value names that are produced. FileCheck `CHECK` lines may contain regex statements as well as named string substitution blocks. Utilizing the above, we end up with the example shown in the main FileCheck tests section.

```
// CHECK-LABEL: func.func @simple_constant
func.func @simple_constant() -> (i32, i32) {
  /// Here we use a substitution variable as the output of the constant is
  /// useful for the test, but we omit as much as possible of everything else.
  // CHECK-NEXT: %[[RESULT:.*]] = arith.constant 1
  // CHECK-NEXT: return %[[RESULT]], %[[RESULT]]

  %0 = arith.constant 1 : i32
  %1 = arith.constant 1 : i32
  return %0, %1 : i32, i32
}
```

Edit on GitHub

← Prev - Glossary Next - Code Documentation →