LLVM Testing Infrastructure Guide

Overview

This document is the reference manual for the LLVM testing infrastructure. It documents the structure of the LLVM testing infrastructure, the tools needed to use it, and how to add and run tests.

Requirements

In order to use the LLVM testing infrastructure, you will need all of the software required to build LLVM, as well as Python 2.7 or later.

If you intend to run the test-suite, you will also need a development version of zlib (zlib1g-dev is known to work on several Linux distributions).

LLVM testing infrastructure organization

The LLVM testing infrastructure contains two major categories of tests: regression tests and whole programs. The regression tests are contained inside the LLVM repository itself under `llvm/test` and are expected to always pass – they should be run before every commit.

The whole programs tests are referred to as the “LLVM test suite” (or “test-suite”) and are in the

`test-suite` module in subversion. For historical reasons, these tests are also referred to as the "nightly tests" in places, which is less ambiguous than "test-suite" and remains in use although we run them much more often than nightly.

Regression tests

The regression tests are small pieces of code that test a specific feature of LLVM or trigger a specific bug in LLVM. The language they are written in depends on the part of LLVM being tested. These tests are driven by the Lit testing tool (which is part of LLVM), and are located in the `llvm/test` directory.

Typically when a bug is found in LLVM, a regression test containing just enough code to reproduce the problem should be written and placed somewhere underneath this directory. For example, it can be a small piece of LLVM IR distilled from an actual application or benchmark.

`test-suite`

The test suite contains whole programs, which are pieces of code which can be compiled and linked into a stand-alone program that can be executed. These programs are generally written in high level languages such as C or C++.

These programs are compiled using a user specified compiler and set of flags, and then executed to capture the program output and timing information. The output of these programs is compared to a reference output to ensure that the program is being compiled correctly.

In addition to compiling and executing programs, whole program tests serve as a way of benchmarking LLVM performance, both in terms of the efficiency of the programs generated as well as the speed with which LLVM compiles, optimizes, and generates code.

The test-suite is located in the `test-suite` Subversion module.

Debugging Information tests

The test suite contains tests to check quality of debugging information. The test are written in C based languages or in LLVM assembly language.

These tests are compiled and run under a debugger. The debugger output is checked to validate of debugging information. See README.txt in the test suite for more information . This test suite is located in the `debuginfo-tests` Subversion module.

Quick start

The tests are located in two separate Subversion modules. The regressions tests are in the main "llvm" module under the directory `llvm/test` (so you get these tests for free with the main LLVM tree). Use `make check-all` to run the regression tests after building LLVM.

The more comprehensive test suite that includes whole programs in C and C++ is in the `test-suite` module. See test-suite Quickstart for more information on running these tests.

Regression tests

To run all of the LLVM regression tests use the check-llvm target:

```
% make check-llvm
```

If you have Clang checked out and built, you can run the LLVM and Clang tests simultaneously using:

```
% make check-all
```

To run the tests with Valgrind (Memcheck by default), use the `LIT_ARGS` make variable to pass the required options to lit. For example, you can use:

```
% make check LIT_ARGS="-v --vg --vg-leak"
```

to enable testing with valgrind and with leak checking enabled.

To run individual tests or subsets of tests, you can use the `llvm-lit` script which is built as part of LLVM. For example, to run the `Integer/BitPacked.ll` test by itself you can run:

```
% llvm-lit ~/llvm/test/Integer/BitPacked.ll
```

or to run all of the ARM CodeGen tests:

```
% llvm-lit ~/llvm/test/CodeGen/ARM
```

For more information on using the **lit** tool, see `llvm-lit --help` or the lit man page.

Debugging Information tests

To run debugging information tests simply checkout the tests inside clang/test directory.

```
% cd clang/test
% svn co http://llvm.org/svn/llvm-project/debuginfo-tests/trunk debuginfo-tests
```

These tests are already set up to run as part of clang regression tests.

Regression test structure

The LLVM regression tests are driven by **lit** and are located in the `llvm/test` directory.

This directory contains a large array of small tests that exercise various features of LLVM and to ensure that regressions do not occur. The directory is broken into several sub-directories, each focused on a particular area of LLVM.

Writing new regression tests

The regression test structure is very simple, but does require some information to be set. This information is gathered via `configure` and is written to a file, `test/lit.site.cfg` in the build directory. The `llvm/test` Makefile does this work for you.

In order for the regression tests to work, each directory of tests must have a `lit.local.cfg` file. **lit** looks for this file to determine how to run the tests. This file is just Python code and thus is very flexible, but we've standardized it for the LLVM regression tests. If you're adding a directory of tests, just copy `lit.local.cfg` from another directory to get running. The standard `lit.local.cfg` simply specifies which files to look in for tests. Any directory that contains only directories does not need the `lit.local.cfg` file. Read the Lit documentation for more information.

Each test file must contain lines starting with "RUN:" that tell **lit** how to run it. If there are no RUN lines, **lit** will issue an error while running a test.

RUN lines are specified in the comments of the test program using the keyword `RUN` followed by a colon, and lastly the command (pipeline) to execute. Together, these lines form the "script" that **lit** executes to run the test case. The syntax of the RUN lines is similar to a shell's syntax for pipelines including I/O redirection and variable substitution. However, even though these lines may *look* like a shell script, they are not. RUN lines are interpreted by **lit**. Consequently, the syntax differs from shell in a few ways. You can specify as many RUN lines as needed.

lit performs substitution on each RUN line to replace LLVM tool names with the full paths to the executable built for each tool (in `$(LLVM_OBJ_ROOT)/$(BuildMode)/bin`). This ensures that **lit** does not invoke any stray LLVM tools in the user's path during testing.

Each RUN line is executed on its own, distinct from other lines unless its last character is `\`. This continuation character causes the RUN line to be concatenated with the next one. In this way you can build up long pipelines of commands without making huge line lengths. The lines ending in `\` are concatenated until a RUN line that doesn't end in `\` is found. This concatenated set of RUN lines then constitutes one execution. **lit** will substitute variables and arrange for the pipeline to be executed. If any process in the pipeline fails, the entire line (and test case) fails too.

Below is an example of legal RUN lines in a `.ll` file:

```
; RUN: llvm-as < %s | llvm-dis > %t1
; RUN: llvm-dis < %s.bc-13 > %t2
; RUN: diff %t1 %t2
```

As with a Unix shell, the RUN lines permit pipelines and I/O redirection to be used.

There are some quoting rules that you must pay attention to when writing your RUN lines. In general nothing needs to be quoted. **lit** won't strip off any quote characters so they will get passed to the invoked program. To avoid this use curly braces to tell **lit** that it should treat everything enclosed as one value.

In general, you should strive to keep your RUN lines as simple as possible, using them only to run tools that generate textual output you can then examine. The recommended way to examine output to figure out if the test passes is using the FileCheck tool. *[The usage of grep in RUN lines is deprecated - please do not send or commit patches that use it.]*

Put related tests into a single file rather than having a separate file per test. Check if there are files already covering your feature and consider adding your code there instead of creating a new file.

Extra files

If your test requires extra files besides the file containing the `RUN:` lines, the idiomatic place to put them is in a subdirectory `Inputs`. You can then refer to the extra files as `%S/Inputs/foo.bar`.

For example, consider `test/Linker/ident.ll`. The directory structure is as follows:

```
test/
  Linker/
    ident.ll
    Inputs/
      ident.a.ll
      ident.b.ll
```

For convenience, these are the contents:

```
;;;;; ident.ll:
```

```
; RUN: llvm-link %S/Inputs/ident.a.ll %S/Inputs/ident.b.ll -S | FileCheck %s

; Verify that multiple input llvm.ident metadata are linked together.

; CHECK-DAG: !llvm.ident = !{!0, !1, !2}
; CHECK-DAG: "Compiler V1"
; CHECK-DAG: "Compiler V2"
; CHECK-DAG: "Compiler V3"

;;;;; Inputs/ident.a.ll:

!llvm.ident = !{!0, !1}
!0 = metadata !{metadata !"Compiler V1"}
!1 = metadata !{metadata !"Compiler V2"}

;;;;; Inputs/ident.b.ll:

!llvm.ident = !{!0}
!0 = metadata !{metadata !"Compiler V3"}
```

For symmetry reasons, `ident.ll` is just a dummy file that doesn't actually participate in the test besides holding the `RUN:` lines.

Note

Some existing tests use `RUN: true` in extra files instead of just putting the extra files in an `Inputs/` directory. This pattern is deprecated.

Fragile tests

It is easy to write a fragile test that would fail spuriously if the tool being tested outputs a full path to the input file. For example, **opt** by default outputs a `ModuleID`:

```
$ cat example.ll
define i32 @main() nounwind {
    ret i32 0
}

$ opt -S /path/to/example.ll
; ModuleID = '/path/to/example.ll'

define i32 @main() nounwind {
    ret i32 0
}
```

`ModuleID` can unexpectedly match against `CHECK` lines. For example:

```
; RUN: opt -S %s | FileCheck

define i32 @main() nounwind {
    ; CHECK-NOT: load
    ret i32 0
}
```

This test will fail if placed into a `download` directory.

To make your tests robust, always use `opt ... < %s` in the RUN line. **opt** does not output a `ModuleID` when input comes from stdin.

Platform-Specific Tests

Whenever adding tests that require the knowledge of a specific platform, either related to code generated, specific output or back-end features, you must make sure to isolate the features, so that buildbots that run on different architectures (and don't even compile all back-ends), don't fail.

The first problem is to check for target-specific output, for example sizes of structures, paths and architecture names, for example:

- Tests containing Windows paths will fail on Linux and vice-versa.
- Tests that check for `x86_64` somewhere in the text will fail anywhere else.
- Tests where the debug information calculates the size of types and structures.

Also, if the test rely on any behaviour that is coded in any back-end, it must go in its own directory. So, for instance, code generator tests for ARM go into `test/CodeGen/ARM` and so on. Those directories contain a special `lit` configuration file that ensure all tests in that directory will only run if a specific back-end is compiled and available.

For instance, on `test/CodeGen/ARM`, the `lit.local.cfg` is:

```
config.suffixes = ['.ll', '.c', '.cpp', '.test']
if not 'ARM' in config.root.targets:
  config.unsupported = True
```

Other platform-specific tests are those that depend on a specific feature of a specific sub-architecture, for example only to Intel chips that support `AVX2`.

For instance, `test/CodeGen/X86/psubus.ll` tests three sub-architecture variants:

```
; RUN: llc -mcpu=core2 < %s | FileCheck %s -check-prefix=SSE2
; RUN: llc -mcpu=corei7-avx < %s | FileCheck %s -check-prefix=AVX1
; RUN: llc -mcpu=core-avx2 < %s | FileCheck %s -check-prefix=AVX2
```

And the checks are different:

```
; SSE2: @test1
; SSE2: psubusw LCPI0_0(%rip), %xmm0
; AVX1: @test1
; AVX1: vpsubusw LCPI0_0(%rip), %xmm0, %xmm0
; AVX2: @test1
; AVX2: vpsubusw LCPI0_0(%rip), %xmm0, %xmm0
```

So, if you're testing for a behaviour that you know is platform-specific or depends on special features of sub-architectures, you must add the specific triple, test with the specific FileCheck and put it into the specific directory that will filter out all other architectures.

Constraining test execution

Some tests can be run only in specific configurations, such as with debug builds or on particular platforms. Use `REQUIRES` and `UNSUPPORTED` to control when the test is enabled.

Some tests are expected to fail. For example, there may be a known bug that the test detect. Use `XFAIL` to mark a test as an expected failure. An `XFAIL` test will be successful if its execution fails, and will be a failure if its execution succeeds.

```
; This test will be only enabled in the build with asserts.
; REQUIRES: asserts
; This test is disabled on Linux.
; UNSUPPORTED: -linux-
; This test is expected to fail on PowerPC.
```

```
; XFAIL: powerpc
```

`REQUIRES` and `UNSUPPORTED` and `XFAIL` all accept a comma-separated list of boolean expressions. The values in each expression may be:

- Features added to `config.available_features` by configuration files such as `lit.cfg`.
- Substrings of the target triple (`UNSUPPORTED` and `XFAIL` only).

`REQUIRES` enables the test if all expressions are true.
`UNSUPPORTED` disables the test if any expression is true.
`XFAIL` expects the test to fail if any expression is true.

As a special case, `XFAIL: *` is expected to fail everywhere.

```
; This test is disabled on Windows,
; and is disabled on Linux, except for Android Linux.
; UNSUPPORTED: windows, linux && !android
; This test is expected to fail on both PowerPC and ARM.
; XFAIL: powerpc || arm
```

Substitutions

Besides replacing LLVM tool names the following substitutions are performed in RUN lines:

`%%`

Replaced by a single `%`. This allows escaping other substitutions.

`%s`

File path to the test case's source. This is suitable for passing on the command line as the input to an LLVM tool.

Example: `/home/user/llvm/test/MC/ELF/foo_test.s`

`%S`

Directory path to the test case's source.

Example: `/home/user/llvm/test/MC/ELF`

`%t`

File path to a temporary file name that could be used for this test case. The file name won't conflict with other test cases. You can append to it if you need multiple temporaries. This is useful as the destination of some redirected output.

Example: `/home/user/llvm.build/test/MC/ELF/Output/foo_test.s.tmp`

`%T`

Directory of `%t`.

Example: `/home/user/llvm.build/test/MC/ELF/Output`

`%{pathsep}`

Expands to the path separator, i.e. `:` (or `;` on Windows).

`%/s, %/S, %/t, %/T:`

Act like the corresponding substitution above but replace any `\` character with a `/`. This is useful to normalize path separators.

Example: `%s:  C:\Desktop Files/foo_test.s.tmp`

Example: `%/s: C:/Desktop Files/foo_test.s.tmp`

`%:s, %:S, %:t, %:T:`

Act like the corresponding substitution above but remove colons at the beginning of Windows paths. This is useful to allow concatenation of absolute paths on Windows to produce a legal path.

Example: `%s:  C:\Desktop Files\foo_test.s.tmp`

Example: `%:s: C\Desktop Files\foo_test.s.tmp`

LLVM-specific substitutions:

`%shlibext`

The suffix for the host platforms shared library files. This includes the period as the first character.

Example: `.so` (Linux), `.dylib` (OS X), `.dll` (Windows)

`%exeext`

The suffix for the host platforms executable files. This includes the period as the first character.

Example: `.exe` (Windows), empty on Linux.

`%(line)`, `%(line+<number>)`, `%(line-<number>)`

The number of the line where this substitution is used, with an optional integer offset. This can be used in tests with multiple RUN lines, which reference test file's line numbers.

Clang-specific substitutions:

`%clang`

Invokes the Clang driver.

`%clang_cpp`

Invokes the Clang driver for C++.

`%clang_cl`

Invokes the CL-compatible Clang driver.

`%clangxx`

Invokes the G++-compatible Clang driver.

`%clang_cc1`

Invokes the Clang frontend.

`%itanium_abi_triple`, `%ms_abi_triple`

These substitutions can be used to get the current target triple adjusted to the desired ABI. For example, if the test suite is running with the `i686-pc-win32` target, `%itanium_abi_triple` will expand to `i686-pc-mingw32`. This allows a test to run with a specific ABI without constraining it to a specific triple.

To add more substituations, look at `test/lit.cfg` or `lit.local.cfg`.

Options

The llvm lit configuration allows to customize some things with user options:

`llc`, `opt`, ...
: Substitute the respective llvm tool name with a custom command line. This allows to specify custom paths and default arguments for these tools. Example:

  % llvm-lit “-Dllc=llc -verify-machineinstrs”

`run_long_tests`
: Enable the execution of long running tests.

`llvm_site_config`
: Load the specified lit configuration instead of the default one.

Other Features

To make RUN line writing easier, there are several helper programs. These helpers are in the PATH when running tests, so you can just call them using their name. For example:

`not`
: This program runs its arguments and then inverts the result code from it. Zero result codes become 1. Non-zero result codes become 0.

To make the output more useful, **lit** will scan the lines of the test case for ones that contain a pattern that matches `PR[0-9]+`. This is the syntax for specifying a PR (Problem Report) number that is related to the test case. The number after “PR” specifies the LLVM bugzilla number. When a PR number is specified, it will be used in the pass/fail reporting. This is useful to quickly get some context when a test fails.

Finally, any line that contains “END.” will cause the special interpretation of lines to terminate. This is generally done right after the last RUN: line. This has two side effects:

a. it prevents special interpretation of lines that are part of the test program, not the instructions to the test case, and
b. it speeds things up for really big test cases by avoiding interpretation of the remainder of the file.

`test-suite` Overview

The `test-suite` module contains a number of programs that can be compiled and executed. The `test-suite` includes reference outputs for all of the programs, so that the output of the executed program can be checked for correctness.

`test-suite` tests are divided into three types of tests: MultiSource, SingleSource, and External.

- `test-suite/SingleSource`

  The SingleSource directory contains test programs that are only a single source file in size. These are usually small benchmark programs or small programs that calculate a particular value. Several such programs are grouped together in each directory.

- `test-suite/MultiSource`

  The MultiSource directory contains subdirectories which contain entire programs with multiple source files. Large benchmarks and whole applications go here.

- `test-suite/External`

The External directory contains Makefiles for building code that is external to (i.e., not distributed with) LLVM. The most prominent members of this directory are the SPEC 95 and SPEC 2000 benchmark suites. The `External` directory does not contain these actual tests, but only the Makefiles that know how to properly compile these programs from somewhere else. When using LNT, use the `--test-externals` option to include these tests in the results.

`test-suite` Quickstart

The modern way of running the `test-suite` is focused on testing and benchmarking complete compilers using the LNT testing infrastructure.

For more information on using LNT to execute the `test-suite`, please see the LNT Quickstart documentation.

`test-suite` Makefiles

Historically, the `test-suite` was executed using a complicated setup of Makefiles. The LNT based approach above is recommended for most users, but there are some testing scenarios which are not supported by the LNT approach. In addition, LNT currently uses the Makefile setup under the covers and so developers who are interested in how LNT works under the hood may want to understand the Makefile based setup.

For more information on the `test-suite` Makefile setup, please see the Test Suite Makefile Guide.