Writing an LLVM Pass: 101

LLVM 2019 tutorial

Andrzej Warzyński

arm

October 2019

Andrzej's Background

arm**DOWNSTREAM**lldbSciComp**Highlander**

Overview

- LLVM pass development crash course
  - Focus on **out-of-tree** development
  - **Linux** and **Mac OS** (with hints for Windows)
  - Focus on the **new pass manager**
  - **Middle-end** passes (IR <-> IR)
- No prior knowledge assumed
  - ... though some familiarity with LLVM and LLVM IR will be helpful
  - I will emphasize the important bits
- The slides contain plenty of links
  - All code examples are available on GitHub: llvm-tutor
- The code was formatted to fit on the slides

Outline

Part 1: Set-up & Background

Part 2: HelloWorld Pass

Part 3: Transformation Pass

Part 4: Analysis Tool

Part 5: Integration With Opt

Part 6: Testing

Part 7: Final hints

Part 1

Set-up & Background

Obtaining LLVM 9

You don't need to build LLVM:

- Mac OS X

```
brew install llvm@9
```

- Ubuntu Bionic

```
wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key | sudo apt-key add -
sudo apt-add-repository "deb http://apt.llvm.org/bionic/ llvm-toolchain-bionic-9.0 main"
sudo apt-get update
sudo apt-get install -y llvm-9 llvm-9-dev llvm-9-tools clang-9
```

- Windows

```
git clone https://github.com/llvm/llvm-project.git
git checkout release/9.x
mkdir build && cd build
cmake -DLLVM_EXPORT_SYMBOLS_FOR_PLUGINS=On -DLLVM_TARGETS_TO_BUILD=X86
  <llvm-project/root/dir>/llvm/
cmake --build .
```

... however, your mileage will vary.

Pass Manager - Legacy vs New

- LLVM has two pass managers:
  - **Legacy Pass Manager** is the default
  - New PM, aka **Pass Manager** can be enabled with **LLVM_USE_NEWPM** CMake variable
- New vs Legacy Pass manager - **previous talks**:
  - ”Passes in LLVM, Part 1”, Ch. Carruth, EuroLLVM 2014, slides, video
  - ”The LLVM Pass Manager Part 2”, Ch. Carruth, LLVM DEVMTG 2014, slides, video
  - ”New PM: taming a custom pipeline of Falcon JIT”, F. Sergeev, EuroLLVM 2018, slides, video
- The official docs are based on the legacy PM
- Implementation based on various C++ patterns and idioms:
  - **C**uriously **R**ecurring **T**emplate **P**attern (Wikipedia)
  - Code re-use through the **Mixin** pattern (blog)
  - **Concept-model** idiom (S. Parent: *”Inheritance is the base class of Evil”*, video)

LLVM Pass - Analysis vs Transformation

- A pass operates on some unit of IR (e.g. Module or Function)
  - Transformation pass will modify it
  - Analysis pass will generate some high-level information
- Analysis results are produced lazily
  - Another pass needs to request the results first
  - Results are cached
  - Analysis manager deals with a non-trivial cache (in)validation problem
- Transformation **pass managers** (e.g. FunctionPassManager) **record what's preserved**
  - Function pass can invalidate Module analysis results, and vice-versa

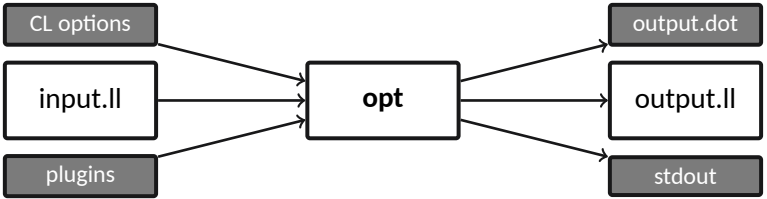
LLVM IR files - function vs module

```
int foo(int a, int b) {
  return a + b;
}
```

```
$ clang -emit-llvm -S -O0 add.c -o add.ll
```

```
; ModuleID = 'add.c'
source_filename = "add.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.14.0"

; Function Attrs: norecurse nounwind readnone ssp uwtable
define i32 @foo(i32, i32) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  ret i32 %3
}

attributes #0 = { norecurse nounwind readnone ssp uwtable "correctly-rounded-divide-sqrt-fp-math"="false" (...) }

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 2, !"SDK Version", [2 x i32] [i32 10, i32 15]}
!1 = !{i32 1, !"wchar_size", i32 4}
!2 = !{i32 7, !"PIC Level", i32 2}
!3 = !{!"Apple clang version 11.0.0 (clang-1100.0.20.17)"}
```

opt



opt workflow

opt, LLVM's modular optimizer, is compiler ninja's best friend:

- takes an LLVM source file (**input.ll**)
  - optimisation - returns an LLVM source file (**output.ll**)
  - analysis - produces analyses results (e.g. to **stdout**)
- load plugins, i.e. shared objects with custom passes
- use **-dot-cfg** to generate CFG files (**output.dot**)

Part 2

HelloWorld Pass

HelloWorld - implementation

llvm-tutor:

```
// Main functionality provided by this pass
void visitor(Function &F) {
  errs() << "Visiting: ";
  errs() << F.getName() << " (takes ";
  errs() << F.arg_size() << " args)\n";
}

struct HelloWorld : llvm::PassInfoMixin<HelloWorld> {
  // Main entry point, takes IR unit to run the
  // pass on (&F) and the corresponding pass
  // manager (to be queried/modified if need be)
  llvm::PreservedAnalyses run(
                        Function &F,
                        FunctionAnalysisManager &) {
    visitor(F);

    // all() is a static method in PreservedAnalyses
    return llvm::PreservedAnalyses::all();
  }
};
```

HelloWorld.cpp

LLVM:

```
template <typename DerivedT> struct PassInfoMixin {
  static StringRef name() {
    // (...)
  }
};

template <typename IRUnitT,
          typename AnalysisManagerT = AnalysisManager<IRUnitT>,
          typename... ExtraArgTs>
class PassManager : public PassInfoMixin<
    PassManager<IRUnitT, AnalysisManagerT, ExtraArgTs...>> {

    PreservedAnalyses run(IRUnitT &IR, AnalysisManagerT &AM,
      ExtraArgTs... ExtraArgs) {
      // Passes is a vector of PassModel<> : PassConcept
      for (unsigned Idx = 0, Size = Passes.size(); Idx != Size; ++Idx) {
        PreservedAnalyses PassPA = P->run(IR, AM, ExtraArgs...);

        AM.invalidate(IR, PassPA);
      }
  } // end of run
} // end of PassManager
```

llvm/include/llvm/IR/PassManager.h

HelloWorld - registration

llvm-tutor:

```
bool FPMHook(StringRef Name, FunctionPassManager &FPM,
             ArrayRef<PassBuilder::PipelineElement>) {
  if (Name != "hello-world")
    return false;

  FPM.addPass(HelloWorld());
  return true;
};

void PBHook(PassBuilder &PB) {
  PB.registerPipelineParsingCallback(FPMHook);
}

llvm::PassPluginLibraryInfo getHelloWorldPluginInfo() {
  return {LLVM_PLUGIN_API_VERSION, "hello-world",
          LLVM_VERSION_STRING, PBHook};
}


// The public entry point for a pass plugin.
extern "C" LLVM_ATTRIBUTE_WEAK llvm::PassPluginLibraryInfo
llvmGetPassPluginInfo() {
  return getHelloWorldPluginInfo();
}
```

HelloWorld.cpp

LLVM:

```
struct PassPluginLibraryInfo {
  /// The API version understood by this plugin
  uint32_t APIVersion;
  /// A meaningful name of the plugin.
  const char *PluginName;
  /// The version of the plugin.
  const char *PluginVersion;

  /// Callback for registering plugin passes with PassBuilder
  void (*RegisterPassBuilderCallbacks)(PassBuilder &);
};
```

include/llvm/Passes/PassPlugin.h

```
// Load requested pass plugins and let them register pass
// builder callbacks
bool runPassPipeline(...) {
                              from CL option
  for (auto &PluginFN :  PassPlugins ) {
    auto PassPlugin = PassPlugin::Load(PluginFN);

                    FPMHook from HelloWorld.cpp
    PassPlugin->registerPassBuilderCallbacks(PB);
  }
}
```

tools/opt/NewPMDriver.cpp

HelloWorld - registration in practice

llvm-tutor:

```
llvm::PassPluginLibraryInfo getHelloWorldPluginInfo() {
  return {LLVM_PLUGIN_API_VERSION, "HelloWorld",
          LLVM_VERSION_STRING, [](PassBuilder &PB) {
            PB.registerPipelineParsingCallback(
                [](StringRef Name, FunctionPassManager &FPM,
                   ArrayRef<PassBuilder::PipelineElement>) {
                  if (Name == "hello-world") {
                    FPM.addPass(HelloWorld());
                    return true;
                  }
                  return false;
                });
          }};
}

// The public entry point for a pass plugin.
extern "C" LLVM_ATTRIBUTE_WEAK llvm::PassPluginLibraryInfo
llvmGetPassPluginInfo() {
  return getHelloWorldPluginInfo();
}
```

HelloWorld.cpp

HelloWorld - CMake

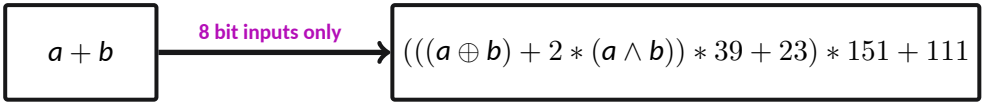
```
# LLVM requires CMake >= 3.4.3
cmake_minimum_required(VERSION 3.4.3)

# Gotcha 1: On Mac OS clang default to C++ 98, LLVM is implemented in C++ 14
set(CMAKE_CXX_STANDARD 14 CACHE STRING "")

# STEP 1. Make sure that LLVMConfig.cmake _is_ on CMake's search patch
set(LT_LLVM_INSTALL_DIR "" CACHE PATH "LLVM installation directory")
set(LT_LLVM_CMAKE_CONFIG_DIR "${LT_LLVM_INSTALL_DIR}/lib/cmake/llvm/")
list(APPEND CMAKE_PREFIX_PATH "${LT_LLVM_CMAKE_CONFIG_DIR}")

# STEP 2. Load LLVM config from ... LLVMConfig.cmake
find_package(LLVM 9.0.0 REQUIRED CONFIG)

# HelloWorld includes header files from LLVM
include_directories(${LLVM_INCLUDE_DIRS})

if(NOT LLVM_ENABLE_RTTI)
  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-rtti")
endif()

# STEP 3. Define the plugin/pass/library
# Gotcha 2: You don't need to use add_llvm_library
add_library(HelloWorld SHARED HelloWorld.cpp)

# Gotcha 3: By default, undefined symbols are not allowed in shared objects on Mac OS. This is expected though so change the behaviour.
target_link_libraries(HelloWorld
  "$<$<PLATFORM_ID:Darwin>:-undefined dynamic_lookup>")
```

CMakeLists.txt

Part 3

Transformation Pass

Transformation pass



Transformation formula

Transformation pass - MBAAdd:

- Replaces 8-bit additions with a sequence of equivalent operations
- Leverages **IRBuilder** and **ReplaceInstWithInst**
- Formula from *"Defeating MBA-based Obfuscation"* N. Eyrolles, L. Goubin, M. Videau

Transformation pass

```
bool MBAAdd::runOnBasicBlock(BasicBlock &BB) {
  bool Changed = false;
  for (auto Inst = BB.begin(), IE = BB.end(); IIT != IE; ++IIT) {
    // Skip non-binary (e.g. unary or compare) instructions
    auto *BinOp = dyn_cast<BinaryOperator>(Inst);
    if (!BinOp)
      continue;

    // Skip instructions other than add
    if (BinOp->getOpcode() != Instruction::Add)
      continue;

    // Skip if the result is not 8-bit wide (this implies that the operands are also 8-bit wide)
    if (!BinOp->getType()->isIntegerTy() || !(BinOp->getType()->getIntegerBitWidth() == 8))
      continue;

    // ... --> go to next slide

    LLVM_DEBUG(dbgs() << *BinOp << " -> " << *NewInst << "\n");
    Changed = true;

    // Update the statistics
    ++SubstCount;
  }
  return Changed;
}
```

MBAAdd.cpp

Transformation pass

```
IRBuilder<> Builder(BinOp);

auto Val39  = ConstantInt::get(BinOp->getType(), 39);
auto Val151 = ConstantInt::get(BinOp->getType(), 151);
auto Val23  = ConstantInt::get(BinOp->getType(), 23);
auto Val2   = ConstantInt::get(BinOp->getType(), 2);
auto Val111 = ConstantInt::get(BinOp->getType(), 111);

Instruction *NewInst =
    BinaryOperator::CreateAdd(
        Val111,
        Builder.CreateMul(
            Val151,
            Builder.CreateAdd(
                Val23,
                Builder.CreateMul(
                    Val39,
                    Builder.CreateAdd(
                        Builder.CreateXor(BinOp->getOperand(0), BinOp->getOperand(1)),
                        Builder.CreateMul(
                            Val2, Builder.CreateAnd(BinOp->getOperand(0), BinOp->getOperand(1))))
                ) // e3 = e2 * 39
            ) // e4 = e2 + 23
        ) // e5 = e4 * 151
    ); // E = e5 + 111
ReplaceInstWithInst(BB.getInstList(), Inst, NewInst);
```

MBAAdd.cpp

Transformation pass

```
#include "llvm/ADT/Statistic.h"
#include "llvm/Support/Debug.h"

#define DEBUG_TYPE "mba-add"

STATISTIC(SubstCount, "The # of substituted instructions");
```

MBAAdd.cpp

- **DEBUG_TYPE** enables **-debug-only=mba-add** in opt (requires debug build)

```
LLVM_DEBUG(dbgs() << *BinOp << " -> " << *NewInst << "\n");
```

  - Alternatively, use **-debug** to print all debug output
- **STATISTIC** enables **-stats** in opt (requires debug build)

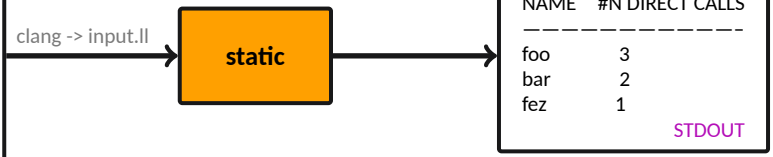
```
++SubstCount;
```

Part 4

Analysis Tool

Analysis Tool - overview

```
void foo() { }
void bar() { foo(); }
void fez() { bar(); }

int main() {
  foo();
  bar();
  fez();

  int ii = 0;
  for (ii = 0; ii < 10; ii++)
    foo();

  return 0;
}
```
input.c



Standalone static analysis tool - static

- **static** is a tool for counting static function calls
- **Note**: in practice we need to compile input.c to **input.ll** first!
- Check CommandLine 2.0 for more on **llvm::cl**

Analysis Tool - StaticCallCounter

```
using ResultStaticCC = llvm::DenseMap<const llvm::Function *, unsigned>;

struct StaticCallCounter : public llvm::AnalysisInfoMixin<StaticCallCounter> {
  using Result = ResultStaticCC;
  Result run(llvm::Module &M, llvm::ModuleAnalysisManager &);
  Result runOnModule(llvm::Module &M);

  // AnalysisKey is a special type used by analysis passes to provide an address that
  // identifies that particular analysis pass type.
  static llvm::AnalysisKey Key;
};
```

StaticCallCounter.cpp

```
template <typename DerivedT>
struct AnalysisInfoMixin : PassInfoMixin<DerivedT> {
  static AnalysisKey *ID() {
    return &DerivedT::Key;
  }
};
```

llvm/include/llvm/IR/PassManager.h

Analysis Tool - StaticCallCounter

```
StaticCallCounter::Result StaticCallCounter::runOnModule(Module &M) {
  llvm::DenseMap<const llvm::Function *, unsigned> Res;

  for (auto &Func : M) {
    for (auto &BB : Func) {
      for (auto &Ins : BB) {
        auto ICS = ImmutableCallSite(&Ins);
        if (nullptr == ICS.getInstruction()) continue; // Skip non-call instructions

        auto DirectInvoc = dyn_cast<Function>(ICS.getCalledValue()->stripPointerCasts());
        if (nullptr == DirectInvoc) continue;         // Skip non-direct function calls

        auto CallCount = Res.find(DirectInvoc);
        if (Res.end() == CallCount) CallCount = Res.insert(std::make_pair(DirectInvoc, 0)).first;

        ++CallCount->second;
      }
    }
  }
  return Res;
}
```

StaticCallCounter.cpp

Analysis Tool - StaticCCWrapper

- **StaticCCWrapper** requests and prints results from StaticCallCounter

```
struct StaticCCWrapper : public PassInfoMixin<StaticCCWrapper> {
  llvm::PreservedAnalyses run(llvm::Module &M, llvm::ModuleAnalysisManager &MAM) {

    ResultStaticCC DirectCalls = MAM.getResult<StaticCallCounter>(M);
    printStaticCCResult(errs(), DirectCalls);

    return llvm::PreservedAnalyses::all();
  }
};
```

- Analysis results can be invalidated like this:

```
    llvm::PreservedAnalyses PA = llvm::PreservedAnalyses::all();
    PA.abandon<StaticCallCounter>();
    return PA;
```

Analysis Tool - main()

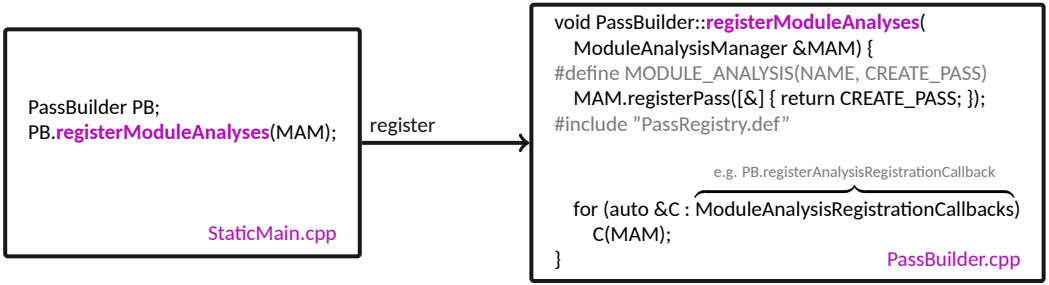
```
static cl::OptionCategory CallCounterCategory{"call counter options"};
static cl::opt<std::string> InputModule{...};

int main(int Argc, char **Argv) {
  cl::HideUnrelatedOptions(CallCounterCategory);
  cl::ParseCommandLineOptions(Argc, Argv, "Counts the number of static function calls in a file \n");
  llvm_shutdown_obj SDO; // Cleans up LLVM objects

  SMDiagnostic Err;
  LLVMContext Ctx;
  std::unique_ptr<Module> M = parseIRFile(InputModule.getValue(), Err, Ctx);

  if (!M) {
    errs() << "Error reading bitcode file: " << InputModule << "\ n";
    Err.print(Argv[0], errs());
    return -1;
  }

  countStaticCalls(*M); // Runs StaticCallCounter
  return 0;
}
```

StaticMain.cpp

Analysis Tool - countStaticCalls()

```
static void countStaticCalls(Module &M) {
  // Create a module pass manager and add StaticCCWrapper to it
  ModulePassManager MPM;
  StaticCCWrapper StaticWrapper;
  MPM.addPass(StaticWrapper);

  // Create an analysis manager and register StaticCallCounter
  ModuleAnalysisManager MAM;
  MAM.registerPass([&] { return StaticCallCounter(); });

  // Register  module analysis passes defined in PassRegistry.def
  PassBuilder PB;
  PB.registerModuleAnalyses(MAM);

  // Finally, run the passes registered with MPM
  MPM.run(M, MAM);
}
```

Analysis Tool - PassBuilder digression



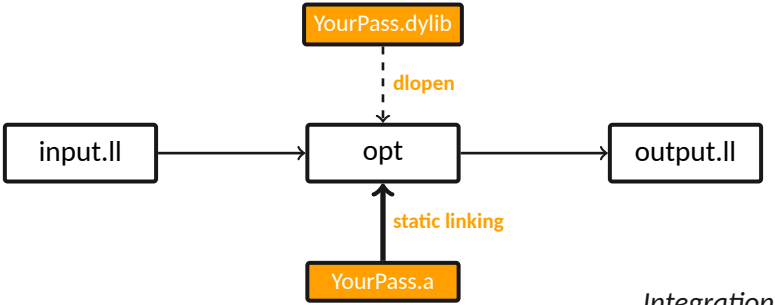
PassBuilder glues everything together:

- Registers all passes from PassRegistry.def (in particular, **PassInstrumentationAnalysis**)
- Registers your plugins via **llvmGetPassPluginInfo()** (recall **HelloWorld** registration)
- Strictly speaking, can be omitted

Part 5

Integration With Opt

Integration With Opt



Integration with opt

Automatic integration with opt/clang/bugpoint:

- Implemented in a patch by **Serge Guelton**, available upstream D61446
- Extends plugin registration for polly

Integration With Opt

- Requires two entry points for the new PM
  - One for static and one for dynamic registration
  - Use **LLVM_BYE_LINK_INTO_TOOLS=ON** for the former

CMake setup:

```
add_llvm_pass_plugin(Bye Bye.cpp)
if (LLVM_LINK_LLVM_DYLIB)
  target_link_libraries(Bye PUBLIC LLVM)
else()
  target_link_libraries(Bye
    PUBLIC
    LLVMSupport
    LLVMCore
    LLVMipo
    LLVMPasses
    )
  endif()
```

llvm/examples/Bye/CMakeLists.txt

Registration:

```
llvm::PassPluginLibraryInfo getByePluginInfo() {
  return {LLVM_PLUGIN_API_VERSION, "Bye", LLVM_VERSION_STRING,
          [](PassBuilder &PB) {
            PB.registerVectorizerStartEPCallback(
                [](llvm::FunctionPassManager &PM,
                   llvm::PassBuilder::OptimizationLevel Level) {
                  PM.addPass(Bye());
                });
          }};
}

#ifndef LLVM_BYE_LINK_INTO_TOOLS
extern "C" LLVM_ATTRIBUTE_WEAK ::llvm::PassPluginLibraryInfo
llvmGetPassPluginInfo() {
  return getByePluginInfo();
}
#endif
```

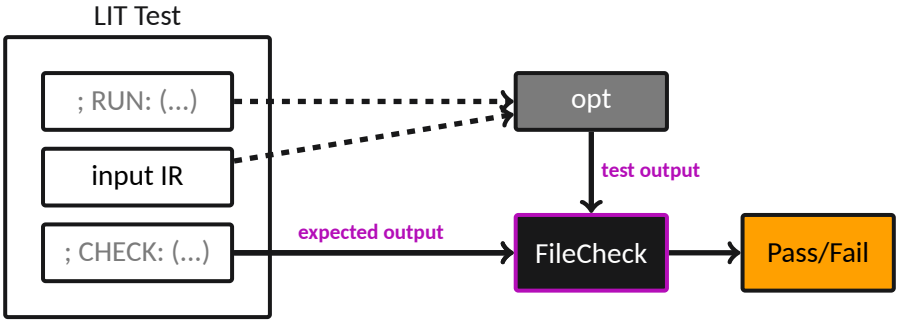
llvm/examples/Bye/Bye.cpp

Part 6

Testing

Testing - LIT Test

LLVM's Integrated Tester



- Every LIT test must contain a **RUN** command:

```
; RUN: opt -load-pass-plugin=../lib/libMBAAdd%shlibext -passes="mba-add" -S %s  | FileCheck %s
```

- LIT test are just regular LLVM files with comments (e.g. ; CHECK: Pattern)
- Tricky part - test/tool discovery

Testing - out of tree LIT set-up

- **llvm-lit** is a Python script that automates test discovery. It:
  - looks for **lit.site.cfg.py** or **lit.cfg** in your test directory
  - defines various global objects that have to be configured locally
- **lit.site.cfg.py** is normally configured by CMake

```
import sys

# config is a global instance of TestingConfig provided by lit
config.llvm_tools_dir = "@LT_LIT_TOOLS_DIR@"
config.llvm_shlib_ext = "@SHLIBEXT@"

import lit.llvm
lit.llvm.initialize(lit_config, config)

config.test_exec_root = os.path.join("@CMAKE_CURRENT_BINARY_DIR@")

# Delegate most of the work to lit.cfg.py
lit_config.load_config(config, "@LT_TEST_SRC_DIR@/lit.cfg.py")
```

lit.site.cfg.py.in

Testing - out of tree LIT set-up

```
import platform
import lit.formats
from lit.llvm import llvm_config # llvm_config is a global instance of LLVMConfig provided by lit
from lit.llvm.subst import ToolSubst

config.name = 'LLVM-TUTOR'
config.test_format = lit.formats.ShTest(not llvm_config.use_lit_shell)
config.test_source_root = os.path.dirname(__file__)
config.suffixes = ['.ll']

if platform.system() == 'Darwin':
    tool_substitutions = [ToolSubst('%clang', "clang", extra_args=["-isysroot", "`xcrun --show-sdk-path`"])
else:
    tool_substitutions = [ToolSubst('%clang', "clang",)]
llvm_config.add_tool_substitutions(tool_substitutions)

tools = ["opt", "FileCheck", "clang"]
llvm_config.add_tool_substitutions(tools, config.llvm_tools_dir)
config.substitutions.append(('%shlibext', config.llvm_shlib_ext))
```

lit.cfg.py

Part 7

Final hints

Fixing problems

- Use **LLVM_DEBUG**, **STATISTICS** or your favourite debugger, e.g. **lldb**:

```
lldb -- $LLVM_DIR//bin/opt -load-pass-plugin lib/libMBAAdd.dylib -passes=mba-add -S MBA_add_32bit.ll
(lldb) b MBAAdd::run
(lldb) r
```

- IR files can be executed directly with **lli**
- **opt** can generate a lot of very useful data:
  - Generate CFG files with **-dot-cfg**
  - Debug pass pipelines with **-debug-pass=Structure** and **-debug-pass=Executions**
- Beware of mixing tools from different directories
  - Use **LLVM**, **clang**, **opt**, **FileCheck** etc from the same location
- Rely on CMake's **find_package** and add sanity-checks in your scripts:

```
set(LT_LLVM_INCLUDE_DIR "$LT_LLVM_INSTALL_DIR/include/llvm")
if(NOT EXISTS "$LT_LLVM_INCLUDE_DIR")
```

References

Big thank you to the amazing LLVM community!

- LLVM IR:
  - *"LLVM IR Tutorial - Phis, GEPs and other things, oh my!"*, V. Bridgers, Felipe de Azevedo Piovezan, EuroLLVM 2019 slides, video
  - *Mapping High Level Constructs to LLVM IR*, Michael Rodler, online book

- LLVM Passes:
  - *"Building, Testing and Debugging a Simple out-of-tree LLVM Pass"*, S. Guelton, A. Guinet, LLVM Dev Meeting 2015 slides, video
  - *"Writing LLVM Pass in 2018"*, blog series by Min-Yih Hsu, link
  - Comments in the implementation of the new pass manager

Happy hacking!

- All examples (and much more):
  https://github.com/banach-space/llvm-tutor/
- @_banach_space, andrzej.warzynski@arm.com