llvm-tutor

Build Status

Example LLVM passes - based on **LLVM 14**

llvm-tutor is a collection of self-contained reference LLVM passes. It's a tutorial that targets novice and aspiring LLVM developers. Key features:

- **Out-of-tree** - builds against a binary LLVM installation (no need to build LLVM from sources)
- **Complete** - includes `CMake` build scripts, LIT tests, CI set-up and documentation
- **Modern** - based on the latest version of LLVM (and updated with every release)

Overview

LLVM implements a very rich, powerful and popular API. However, like many complex technologies, it can be quite daunting and overwhelming to learn and master. The goal of this LLVM tutorial is to showcase that LLVM can in fact be easy and fun to work with. This is demonstrated through a range self-contained, testable LLVM passes, which are implemented using idiomatic LLVM.

This document explains how to set-up your environment, build and run the examples, and go about debugging. It contains a high-level overview of the implemented examples and contains some background information on writing LLVM passes. The source files, apart from the code itself, contain comments that will guide you through the implementation. All examples are complemented with LIT tests and reference input files.

Visit **clang-tutor** if you are internested in similar tutorial for Clang.

Table of Contents

HelloWorld: Your First Pass

The **HelloWorld** pass from
HelloWorld.cpp
is a self-contained *reference example*. The corresponding
CMakeLists.txt
implements the minimum set-up for an out-of-source pass.

For every function defined in the input module, **HelloWorld** prints its name
and the number of arguments that it takes. You can build it like this:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
mkdir build
cd build
cmake -DLT_LLVM_INSTALL_DIR=$LLVM_DIR <source/dir/llvm/tutor>/HelloWorld/
make
```

Before you can test it, you need to prepare an input file:

```
# Generate an LLVM test file
$LLVM_DIR/bin/clang -O1 -S -emit-llvm <source/dir/llvm/tutor>/inputs/input_for_hello.c -o
```

Finally, run **HelloWorld** with
opt (use `libHelloWorld.so`
on Linux and `libHelloWorld.dylib` on Mac OS):

```
# Run the pass
$LLVM_DIR/bin/opt -load-pass-plugin ./libHelloWorld.{so|dylib} -passes=hello-world -disabl
# Expected output
(llvm-tutor) Hello from: foo
(llvm-tutor)   number of arguments: 1
(llvm-tutor) Hello from: bar
(llvm-tutor)   number of arguments: 2
(llvm-tutor) Hello from: fez
(llvm-tutor)   number of arguments: 3
(llvm-tutor) Hello from: main
(llvm-tutor)   number of arguments: 2
```

The **HelloWorld** pass doesn't modify the input module. The `-disable-output`
flag is used to prevent **opt** from printing the output bitcode file.

Development Environment

Platform Support And Requirements

This project has been tested on **Ubuntu 20.04** and **Mac OS X 10.14.4**. In
order to build **llvm-tutor** you will need:

- LLVM 14
- C++ compiler that supports C++14
- CMake 3.13.4 or higher

In order to run the passes, you will need:

- **clang-14** (to generate input LLVM files)
- **opt** (to run the passes)

There are additional requirements for tests (these will be satisfied by
installing LLVM 14):

- **lit** (aka **llvm-lit**,
  LLVM tool for executing the tests)
- **FileCheck** (LIT
  requirement, it's used to check whether tests generate the expected output)

Installing LLVM 14 on Mac OS X

On Darwin you can install LLVM 14 with Homebrew:

```
brew install llvm@14
```

If you already have an older version of LLVM installed, you can upgrade it to
LLVM 14 like this:

```
brew upgrade llvm
```

Once the installation (or upgrade) is complete, all the required header files,
libraries and tools will be located in `/usr/local/opt/llvm/`.

Installing LLVM 14 on Ubuntu

On Ubuntu Bionic, you can install modern
LLVM
from the official repository:

```
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-14 main"
sudo apt-get update
sudo apt-get install -y llvm-14 llvm-14-dev llvm-14-tools clang-14
```

This will install all the required header files, libraries and tools in
`/usr/lib/llvm-14/`.

Building LLVM 14 From Sources

Building from sources can be slow and tricky to debug. It is not necessary, but might be your preferred way of obtaining LLVM 14. The following steps will work on Linux and Mac OS X:

```
git clone https://github.com/llvm/llvm-project.git
cd llvm-project
git checkout release/14.x
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release -DLLVM_TARGETS_TO_BUILD=host -DLLVM_ENABLE_PROJECTS=clang
cmake --build .
```

For more details read the official documentation.

Building & Testing

Building

You can build **llvm-tutor** (and all the provided pass plugins) as follows:

```
cd <build/dir>
cmake -DLT_LLVM_INSTALL_DIR=<installation/dir/of/llvm/14> <source/dir/llvm/tutor>
make
```

The `LT_LLVM_INSTALL_DIR` variable should be set to the root of either the installation or build directory of LLVM 14. It is used to locate the corresponding `LLVMConfig.cmake` script that is used to set the include and library paths.

Testing

In order to run **llvm-tutor** tests, you need to install **llvm-lit** (aka

lit). It's not bundled with LLVM 14 packages, but you can install it with **pip**:

```
# Install lit - note that this installs lit globally
pip install lit
```

Running the tests is as simple as:

```
$ lit <build_dir>/test
```

Voilà! You should see all tests passing.

LLVM Plugins as shared objecs

In **llvm-tutor** every LLVM pass is implemented in a separate shared object (you can learn more about shared objects here).
These shared objects are essentially dynamically loadable plugins for **opt**.
All plugins are built in the `<build/dir>/lib` directory.

Note that the extension of dynamically loaded shared objects differs between Linux and Mac OS. For example, for the **HelloWorld** pass you will get:

- `libHelloWorld.so` on Linux
- `libHelloWorld.dylib` on MacOS.

For the sake of consistency, in this README.md file all examples use the `*.so` extension. When working on Mac OS, use `*.dylib` instead.

Overview of The Passes

The available passes are categorised as either Analysis, Transformation or CFG. The difference between Analysis and Transformation passes is rather self-explanatory (here is a more technical breakdown). A CFG pass is simply a Transformation pass that modifies the Control

Flow Graph. This is frequently a bit more complex and requires some extra bookkeeping, hence a dedicated category.

In the following table the passes are grouped thematically and ordered by the level of complexity.

Name	Description	Category
HelloWorld	visits all functions and prints their names	Analysis
OpcodeCounter	prints a summary of LLVM IR opcodes in the input module	Analysis
InjectFuncCall	instruments the input module by inserting calls to `printf`	Transformation
StaticCallCounter	counts direct function calls at compile-time (static analysis)	Analysis
DynamicCallCounter	counts direct function calls at run-time (dynamic analysis)	Transformation
MBASub	obfuscate integer `sub` instructions	Transformation
MBAAdd	obfuscate 8-bit integer `add` instructions	Transformation
FindFCmpEq	finds floating-point equality comparisons	Analysis
ConvertFCmpEq	converts direct floating-point equality comparisons to difference comparisons	Transformation
RIV	finds reachable integer values for each basic block	Analysis
DuplicateBB	duplicates basic blocks, requires **RIV** analysis results	CFG
MergeBB	merges duplicated basic blocks	CFG

Once you've built this project, you can experiment with every pass separately. All passes, except for **HelloWorld**, are described in more details

below.

LLVM passes work with LLVM IR files. You can generate one like this:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
# Textual form
$LLVM_DIR/bin/clang -O1 -emit-llvm input.c -S -o out.ll
# Binary/bit-code form
$LLVM_DIR/bin/clang -O1 -emit-llvm input.c -c -o out.bc
```

It doesn't matter whether you choose the binary, `*.bc` (default), or textual/LLVM assembly form (`.ll` , requires the `-S` flag). Obviously, the latter is more human-readable. Similar logic applies to **opt** - by default it generates `*.bc` files. You can use `-S` to have the output written as `*.ll` files instead.

Note that `clang` adds the `optnone` function attribute if either

- no optimization level is specified, or
- `-O0` is specified.

If you want to compile at `-O0` , you need to specify `-O0 -Xclang -disable-O0-optnone` or define a static isRequired method in your pass. Alternatively, you can specify `-O1` or higher. Otherwise the new pass manager will register the pass but your pass will not be executed.

As noted earlier, all examples in this file use the `*.so` extension for pass plugins. When working on Mac OS, use `*.dylib` instead.

OpcodeCounter

OpcodeCounter is an Analysis pass that prints a summary of the LLVM IR opcodes encountered in every function in the input module. This pass can be run

automatically with one of the
pre-defined optimisation pipelines. However, let's use our tried and tested method
first.

Run the pass

We will use
input_for_cc.c
to test **OpcodeCounter**. Since **OpcodeCounter** is an Analysis pass, we want
opt to print its results. There are two ways of achieving this. First, you
need to choose which pass manager you want to use (see
here for more details). Next:

- Legacy Pass Manager: use the `-analyze` command line option. This option is used to instruct **opt** to print the results of the analysis pass that has just been run.
- New Pass Manager: Simply use the printing pass that corresponds to **OpcodeCounter**. This pass is called `print<opcode-counter>`. No extra arguments are needed, but it's a good idea to add `-disable-output` (it is not required when using `-analyze`).

```
export LLVM_DIR=<installation/dir/of/llvm/14>
# Generate an LLVM file to analyze
$LLVM_DIR/bin/clang -emit-llvm -c <source_dir>/inputs/input_for_cc.c -o input_for_cc.bc
# Run the pass through opt - Legacy PM
$LLVM_DIR/bin/opt -enable-new-pm=0 -load <build_dir>/lib/libOpcodeCounter.so -legacy-opcod
# Run the pass through opt - New PM
$LLVM_DIR/bin/opt -load-pass-plugin <build_dir>/lib/libOpcodeCounter.so --passes="print<op
```

For `main`, **OpcodeCounter** prints the following summary (note that when running the pass,
a summary for other functions defined in `input_for_cc.bc` is also printed):

```
================================================
LLVM-TUTOR: OpcodeCounter results for `main`
================================================
OPCODE               #N TIMES USED
------------------------------------------------
load                 2
br                   4
icmp                 1
add                  1
ret                  1
alloca               2
store                4
call                 4
------------------------------------------------
```

Auto-registration with optimisation pipelines

You can run **OpcodeCounter** by simply specifying an optimisation level (e.g. `-O{1|2|3|s}`). This is achieved through auto-registration with the existing optimisation pass pipelines. Note that you still have to specify the plugin file to be loaded:

```
$LLVM_DIR/bin/opt -load <build_dir>/lib/libOpcodeCounter.so -O1 input_for_cc.bc
```

In this example I used the Legacy Pass Manager (the plugin file was specified with `-load` rather than `-load-pass-plugin`). The auto-registration also works with the New Pass Manager:

```
$LLVM_DIR/bin/opt -load-pass-plugin <build_dir>/lib/libOpcodeCounter.so --passes='default<
```

This is implemented in
OpcodeCounter.cpp,
on
line 122 for the New PM, and on
line 159 for the Legacy PM.
This section contains more information about
the pass managers in LLVM.

InjectFuncCall

This pass is a *HelloWorld* example for *code instrumentation*. For every function defined in the input module, **InjectFuncCall** will add (*inject*) the following call to `printf`:

```
printf("(llvm-tutor) Hello from: %s\n(llvm-tutor)   number of arguments: %d\n", FuncName,
```

This call is added at the beginning of each function (i.e. before any other instruction). `FuncName` is the name of the function and `FuncNumArgs` is the number of arguments that the function takes.

Run the pass

We will use
input_for_hello.c
to test **InjectFuncCall**:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
# Generate an LLVM file to analyze
$LLVM_DIR/bin/clang -O0 -emit-llvm -c <source_dir>/inputs/input_for_hello.c -o input_for_h
# Run the pass through opt - Legacy PM
$LLVM_DIR/bin/opt -enable-new-pm=0 -load <build_dir>/lib/libInjectFuncCall.so -legacy-inje
# Run the pass through opt - New PM
$LLVM_DIR/bin/opt -load-pass-plugin <build_dir>/lib/libInjectFuncCall.so --passes="inject-
```

This generates `instrumented.bin`, which is the instrumented version of `input_for_hello.bc`. In order to verify that **InjectFuncCall** worked as expected, you can either check the output file (and verify that it contains extra calls to `printf`) or run it:

```
$LLVM_DIR/bin/lli instrumented.bin
(llvm-tutor) Hello from: main
(llvm-tutor)   number of arguments: 2
(llvm-tutor) Hello from: foo
(llvm-tutor)   number of arguments: 1
(llvm-tutor) Hello from: bar
(llvm-tutor)   number of arguments: 2
(llvm-tutor) Hello from: foo
(llvm-tutor)   number of arguments: 1
(llvm-tutor) Hello from: fez
(llvm-tutor)   number of arguments: 3
(llvm-tutor) Hello from: bar
(llvm-tutor)   number of arguments: 2
(llvm-tutor) Hello from: foo
(llvm-tutor)   number of arguments: 1
```

InjectFuncCall vs HelloWorld

You might have noticed that **InjectFuncCall** is somewhat similar to **HelloWorld**. In both cases the pass visits all functions, prints their names and the number of arguments. The difference between the two passes becomes quite apparent when you compare the output generated for the same input file, e.g. `input_for_hello.c`. The number of times `Hello from` is printed is either:

- once per every function call in the case of **InjectFuncCall**, or
- once per function definition in the case of **HelloWorld**.

This makes perfect sense and hints how different the two passes are. Whether to print `Hello from` is determined at either:

- run-time for **InjectFuncCall**, or
- compile-time for **HelloWorld**.

Also, note that in the case of **InjectFuncCall** we had to first run the pass with **opt** and then execute the instrumented IR module in order to see the output. For **HelloWorld** it was sufficient to run run the pass with **opt**.

StaticCallCounter

The **StaticCallCounter** pass counts the number of *static* function calls in the input LLVM module. *Static* refers to the fact that these function calls are compile-time calls (i.e. visible during the compilation). This is in contrast to *dynamic* function calls, i.e. function calls encountered at run-time (when the compiled module is run). The distinction becomes apparent when analysing functions calls within loops, e.g.:

```
  for (i = 0; i < 10; i++)
    foo();
```

Although at run-time `foo` will be executed 10 times, **StaticCallCounter** will report only 1 function call.

This pass will only consider direct functions calls. Functions calls via function pointers are not taken into account.

Run the pass through opt

We will use
input_for_cc.c
to test **StaticCallCounter**:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
# Generate an LLVM file to analyze
$LLVM_DIR/bin/clang -emit-llvm -c <source_dir>/inputs/input_for_cc.c -o input_for_cc.bc
# Run the pass through opt - Legacy PM
$LLVM_DIR/bin/opt -enable-new-pm=0 -load <build_dir>/lib/libStaticCallCounter.so -legacy-s
```

You should see the following output:

```
================================================
LLVM-TUTOR: static analysis results
================================================
NAME                 #N DIRECT CALLS
------------------------------------------------
foo                  3
bar                  2
fez                  1
------------------------------------------------
```

Note the extra command line option above: `-analyze`. It's required to inform **opt** to print the results of the analysis to `stdout`. We discussed this option in more detail here.

Run the pass through `static`

You can run **StaticCallCounter** through a standalone tool called `static`.
`static` is an LLVM based tool implemented in StaticMain.cpp.
It is a command line wrapper that allows you to run **StaticCallCounter** without the need for **opt**:

```
<build_dir>/bin/static input_for_cc.bc
```

It is an example of a relatively basic static analysis tool. Its implementation demonstrates how basic pass management in LLVM works (i.e. it handles that for itself instead of relying on **opt**).

DynamicCallCounter

The **DynamicCallCounter** pass counts the number of *run-time* (i.e. encountered during the execution) function calls. It does so by inserting call-counting instructions that are executed every time a function is called. Only calls to functions that are *defined* in the input module are counted.
This pass builds on top of ideas presented in **InjectFuncCall**. You may want to experiment with that example first.

Run the pass

We will use
input_for_cc.c
to test **DynamicCallCounter**:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
# Generate an LLVM file to analyze
$LLVM_DIR/bin/clang -emit-llvm -c <source_dir>/inputs/input_for_cc.c -o input_for_cc.bc
# Instrument the input file
$LLVM_DIR/bin/opt -enable-new-pm=0 -load <build_dir>/lib/libDynamicCallCounter.so -legacy-
```

This generates `instrumented.bin`, which is the instrumented version of `input_for_cc.bc`. In order to verify that **DynamicCallCounter** worked as expected, you can either check the output file (and verify that it contains new call-counting instructions) or run it:

```
# Run the instrumented binary
$LLVM_DIR/bin/lli -jit-kind=mcjit ./instrumented_bin
```

You will see the following output:

```
================================================
LLVM-TUTOR: dynamic analysis results
================================================
NAME                 #N DIRECT CALLS
------------------------------------------------
foo                  13
bar                  2
fez                  1
main                 1
```

DynamicCallCounter vs StaticCallCounter

The number of function calls reported by **DynamicCallCounter** and **StaticCallCounter** are different, but both results are correct. They correspond to *run-time* and *compile-time* function calls respectively. Note also that for **StaticCallCounter** it was sufficient to run the pass through **opt** to have the summary printed. For **DynamicCallCounter** we had to *run*

the instrumented binary to see the output. This is similar to what we observed when comparing HelloWorld and InjectFuncCall.

Mixed Boolean Arithmetic Transformations

These passes implement mixed boolean arithmetic transformations. Similar transformation are often used in code obfuscation (you may also know them from Hacker's Delight) and are a great illustration of what and how LLVM passes can be used for.

Similar transformation are possible at the source-code level. The relevant Clang plugins are available in **clang-tutor**.

MBASub

The **MBASub** pass implements this rather basic expression:

```
a - b == (a + ~b) + 1
```

Basically, it replaces all instances of integer `sub` according to the above formula. The corresponding LIT tests verify that both the formula and that the implementation are correct.

Run the pass

We will use input_for_mba_sub.c to test **MBASub**:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
$LLVM_DIR/bin/clang -emit-llvm -S <source_dir>/inputs/input_for_mba_sub.c -o input_for_sub
$LLVM_DIR/bin/opt -load <build_dir>/lib/libMBASub.so -legacy-mba-sub -S input_for_sub.ll -
```

MBAAdd

The **MBAAdd** pass implements a slightly more involved formula that is only valid for 8 bit integers:

```
a + b == (((a ^ b) + 2 * (a & b)) * 39 + 23) * 151 + 111
```

Similarly to `MBASub`, it replaces all instances of integer `add` according to the above identity, but only for 8-bit integers. The LIT tests verify that both the formula and the implementation are correct.

Run the pass

We will use
input_for_add.c
to test **MBAAdd**:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
$LLVM_DIR/bin/clang -O1 -emit-llvm -S <source_dir>/inputs/input_for_mba.c -o input_for_mba
$LLVM_DIR/bin/opt -load <build_dir>/lib/libMBAAdd.so -legacy-mba-add -S input_for_mba.ll -
```

You can also specify the level of *obfuscation* on a scale of `0.0` to `1.0`, with `0` corresponding to no obfuscation and `1` meaning that all `add` instructions are to be replaced with `(((a ^ b) + 2 * (a & b)) * 39 + 23) * 151 + 111`, e.g.:

```
$LLVM_DIR/bin/opt -load <build_dir>/lib/libMBAAdd.so -legacy-mba-add -mba-ratio=0.3 <sourc
```

RIV

RIV is an analysis pass that for each basic
block BB in
the input function computes the set reachable integer values, i.e. the integer values that are visible (i.e. can be used) in BB. Since the pass operates on the LLVM IR representation of the input file, it takes into account all values that have integer type in
the LLVM IR sense. In particular, since

at the LLVM IR level booleans are represented as 1-bit wide integers (i.e. `i1`), you will notice that booleans are also included in the result.

This pass demonstrates how to request results from other analysis passes in LLVM. In particular, it relies on the Dominator Tree analysis pass from LLVM, which is is used to obtain the dominance tree for the basic blocks in the input function.

Run the pass

We will use input_for_riv.c to test **RIV**:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
# Generate an LLVM file to analyze
$LLVM_DIR/bin/clang -emit-llvm -S -O1 <source_dir>/inputs/input_for_riv.c -o input_for_riv
# Run the pass through opt - Legacy PM
$LLVM_DIR/bin/opt -enable-new-pm=0 -load <build_dir>/lib/libRIV.so -legacy-riv -analyze in
```

You will see the following output:

```
================================================
LLVM-TUTOR: RIV analysis results
================================================
BB id      Reachable Ineger Values
------------------------------------------------
BB %entry
              i32 %a
              i32 %b
              i32 %c
BB %if.then
                %add = add nsw i32 %a, 123
                %cmp = icmp sgt i32 %a, 0
              i32 %a
              i32 %b
              i32 %c
BB %if.end8
                %add = add nsw i32 %a, 123
                %cmp = icmp sgt i32 %a, 0
              i32 %a
              i32 %b
              i32 %c
BB %if.then2
                %mul = mul nsw i32 %b, %a
                %div = sdiv i32 %b, %c
                %cmp1 = icmp eq i32 %mul, %div
                %add = add nsw i32 %a, 123
                %cmp = icmp sgt i32 %a, 0
              i32 %a
              i32 %b
              i32 %c
BB %if.else
                %mul = mul nsw i32 %b, %a
                %div = sdiv i32 %b, %c
                %cmp1 = icmp eq i32 %mul, %div
                %add = add nsw i32 %a, 123
                %cmp = icmp sgt i32 %a, 0
              i32 %a
              i32 %b
              i32 %c
```

Note the extra command line option above: `-analyze`. It's required to inform **opt** to print the results of the analysis to `stdout`. We discussed this option in more detail here.

DuplicateBB

This pass will duplicate all basic blocks in a module, with the exception of basic blocks for which there are no reachable integer values (identified through the **RIV** pass). An example of such a basic block is the entry block in a function that:

- takes no arguments and
- is embedded in a module that defines no global values.

Basic blocks are duplicated by first inserting an `if-then-else` construct and then cloning all the instructions from the original basic block (with the exception of PHI nodes) into two new basic blocks (clones of the original basic block). The `if-then-else` construct is introduced as a non-trivial mechanism that decides which of the cloned basic blocks to branch to. This condition is equivalent to:

```
if (var == 0)
  goto clone 1
else
  goto clone 2
```

in which:

- `var` is a randomly picked variable from the `RIV` set for the current basic block
- `clone 1` and `clone 2` are labels for the cloned basic blocks.

The complete transformation looks like this:

```
BEFORE:                     AFTER:
-------                     ------
                              [ if-then-else ]
             DuplicateBB           /  \
[ BB ]      ------------>   [clone 1] [clone 2]
                                  \  /
                                [ tail ]

LEGEND:
-------
[BB]           - the original basic block
[if-then-else] - a new basic block that contains the if-then-else statement (inserted by D
[clone 1|2]    - two new basic blocks that are clones of BB (inserted by DuplicateBB)
[tail]         - the new basic block that merges [clone 1] and [clone 2] (inserted by Dupl
```

As depicted above, **DuplicateBB** replaces qualifying basic blocks with 4 new basic blocks. This is implemented through LLVM's SplitBlockAndInsertIfThenElse.
DuplicateBB does all the necessary preparation and clean-up. In other words, it's an elaborate wrapper for LLVM's `SplitBlockAndInsertIfThenElse`.

Run the pass

This pass depends on the **RIV** pass, which also needs be loaded in order for **DuplicateBB** to work. Lets use input_for_duplicate_bb.c as our sample input. First, generate the LLVM file:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
$LLVM_DIR/bin/clang -emit-llvm -S -O1 <source_dir>/inputs/input_for_duplicate_bb.c -o inpu
```

Function `foo` in `input_for_duplicate_bb.ll` should look like this (all metadata has been stripped):

```
define i32 @foo(i32) {
  ret i32 1
}
```

Note that there's only one basic block (the *entry* block) and that `foo` takes one argument (this means that the result from **RIV** will be a non-empty set).

We will now apply **DuplicateBB** to `foo`:

```
$LLVM_DIR/bin/opt -load <build_dir>/lib/libRIV.so -load <build_dir>/lib/libDuplicateBB.so
```

After the instrumentation `foo` will look like this (all metadata has been stripped):

```
define i32 @foo(i32) {
lt-if-then-else-0:
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %lt-if-then-0, label %lt-else-0

clone-1-0:
  br label %lt-tail-0

clone-2-0:
  br label %lt-tail-0

lt-tail-0:
  ret i32 1
}
```

There are four basic blocks instead of one. All new basic blocks end with a numeric id of the original basic block (`0` in this case). `lt-if-then-else-0` contains the new `if-then-else` condition. `clone-1-0` and `clone-2-0` are clones of the original basic block in `foo`. `lt-tail-0` is the extra basic block that's required to merge `clone-1-0` and `clone-2-0`.

MergeBB

MergeBB will merge qualifying basic blocks that are identical. To some extent, this pass reverts the transformations introduced by **DuplicateBB**. This is illustrated below:

```
BEFORE:                    AFTER DuplicateBB:                AFTER MergeBB:
-------                    ------------------                --------------
                             [ if-then-else ]                [ if-then-else* ]
             DuplicateBB          /  \              MergeBB         |
[ BB ]      ------------>   [clone 1] [clone 2]     -------->    [ clone ]
                                 \  /                             |
                              [ tail ]                         [ tail* ]

LEGEND:
-------
[BB]           - the original basic block
[if-then-else] - a new basic block that contains the if-then-else statement (**DuplicateBB
[clone 1|2]    - two new basic blocks that are clones of BB (**DuplicateBB**)
[tail]         - the new basic block that merges [clone 1] and [clone 2] (**DuplicateBB**)
[clone]        - [clone 1] and [clone 2] after merging, this block should be very similar
[label*]       - [label] after being updated by **MergeBB**
```

Recall that **DuplicateBB** replaces all qualifying basic block with four new basic blocks, two of which are clones of the original block. **MergeBB** will merge those two clones back together, but it will not remove the remaining two blocks added by **DuplicateBB** (it will update them though).

Run the pass

Lets use the following IR implementation of `foo` as input. Note that basic blocks 3 and 5 are identical and can safely be merged:

```
define i32 @foo(i32) {
  %2 = icmp eq i32 %0, 19
  br i1 %2, label %3, label %5

; <label>:3:
  %4 = add i32 %0,  13
  br label %7

; <label>:5:
  %6 = add i32 %0,  13
  br label %7

; <label>:7:
  %8 = phi i32 [ %4, %3 ], [ %6, %5 ]
  ret i32 %8
}
```

We will now apply **MergeBB** to `foo`:

```
$LLVM_DIR/bin/opt -load <build_dir>/lib/libMergeBB.so -legacy-merge-bb -S foo.ll -o merge.
```

After the instrumentation `foo` will look like this (all metadata has been stripped):

```
define i32 @foo(i32) {
  %2 = icmp eq i32 %0, 19
  br i1 %2, label %3, label %3

3:
  %4 = add i32 %0, 13
  br label %5

5:
  ret i32 %4
}
```

As you can see, basic blocks 3 and 5 from the input module have been merged into one basic block.

Run MergeBB on the output from DuplicateBB

It is really interesting to see the effect of **MergeBB** on the output from **DuplicateBB**. Lets start with the same input as we used for **DuplicateBB**:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
$LLVM_DIR/bin/clang -emit-llvm -S -O1 <source_dir>/inputs/input_for_duplicate_bb.c -o inpu
```

Now we will apply **DuplicateBB** *and* **MergeBB** (in this order) to `foo`.
Recall that **DuplicateBB** requires **RIV**, which means that in total we have to load three plugins:

```
$LLVM_DIR/bin/opt -load-pass-plugin <build_dir>/lib/libRIV.so -load-pass-plugin <build_dir
```

And here's the output:

```
define i32 @foo(i32) {
lt-if-then-else-0:
  %1 = icmp eq i32 %0, 0
  br i1 %1, label %lt-clone-2-0, label %lt-clone-2-0

lt-clone-2-0:
  br label %lt-tail-0

lt-tail-0:
  ret i32 1
}
```

Compare this with the output generated by **DuplicateBB**.
Only one of the clones, `lt-clone-2-0`, has been preserved, and `lt-if-then-else-0` has been updated accordingly. Regardless of the value of of the `if` condition (more precisely, variable `%1`), the control flow jumps to `lt-clone-2-0`.

FindFCmpEq

The **FindFCmpEq** pass finds all floating-point comparison operations that directly check for equality between two values. This is important because these sorts of comparisons can sometimes be indicators of logical issues due to rounding errors inherent in floating-point arithmetic.

FindFCmpEq is implemented as two passes: an analysis pass (`FindFCmpEq`) and a printing pass (`FindFCmpEqPrinter`). The legacy implementation (`FindFCmpEqWrapper`) makes use of both of these passes.

Run the pass

We will use input_for_fcmp_eq.ll
to test **FindFCmpEq**:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
# Generate the input file
$LLVM_DIR/bin/clang -emit-llvm -S -c <source_dir>/inputs/input_for_fcmp_eq.c -o input_for_
# Run the pass
$LLVM_DIR/bin/opt --load-pass-plugin <build_dir>/lib/libFindFCmpEq.so --passes="print<find
```

For the legacy implementation, the `opt` command would be changed to the following:

```
$LLVM_DIR/bin/opt -load <build_dir>/lib/libFindFCmpEq.so -find-fcmp-eq -analyze input_for_
```

In either case, you should see the following output which lists the direct floating-point equality comparison instructions found:

```
Floating-point equality comparisons in "sqrt_impl":
  %cmp = fcmp oeq double %0, %1
Floating-point equality comparisons in "compare_fp_values":
  %cmp = fcmp oeq double %0, %1
```

ConvertFCmpEq

The **ConvertFCmpEq** pass is a transformation that uses the analysis results of **FindFCmpEq** to convert direct floating-point equality comparison instructions into logically equivalent ones that use a pre-calculated rounding threshold.

Run the pass

As with **FindFCmpEq**, we will use input_for_fcmp_eq.ll to test **ConvertFCmpEq**:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
$LLVM_DIR/bin/clang -emit-llvm -S -Xclang -disable-O0-optnone \
  -c <source_dir>/inputs/input_for_fcmp_eq.c -o input_for_fcmp_eq.ll
$LLVM_DIR/bin/opt --load-pass-plugin <build_dir>/lib/libFindFCmpEq.so \
  --load-pass-plugin <build_dir>/lib/libConvertFCmpEq.so \
  --passes=convert-fcmp-eq -S input_for_fcmp_eq.ll -o fcmp_eq_after_conversion.ll
```

For the legacy implementation, the `opt` command would be changed to the following:

```
$LLVM_DIR/bin/opt -load <build_dir>/lib/libFindFCmpEq.so \
  <build_dir>/lib/libConvertFCmpEq.so -convert-fcmp-eq \
  -S input_for_fcmp_eq.ll -o fcmp_eq_after_conversion.ll
```

Notice that both `libFindFCmpEq.so` *and* `libConvertFCmpEq.so` must be loaded -- and the load order matters. Since **ConvertFCmpEq** requires **FindFCmpEq**, its library must be loaded before **ConvertFCmpEq**. If both passes were built as part of the same library, this would not be required.

After transformation, both `fcmp oeq` instructions will have been converted to difference based `fcmp olt` instructions using the IEEE 754 double-precision machine epsilon constant as the round-off threshold:

```
  %cmp = fcmp oeq double %0, %1
```

... has now become

```
  %3 = fsub double %0, %1
  %4 = bitcast double %3 to i64
  %5 = and i64 %4, 9223372036854775807
  %6 = bitcast i64 %5 to double
  %cmp = fcmp olt double %6, 0x3CB0000000000000
```

The values are subtracted from each other and the absolute value of their difference is calculated. If this absolute difference is less than the value of the machine epsilon, the original two floating-point values are considered to be equal.

Debugging

Before running a debugger, you may want to analyze the output from
LLVM_DEBUG
and

STATISTIC

macros. For example, for **MBAAdd**:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
$LLVM_DIR/bin/clang -emit-llvm -S -O1 <source_dir>/inputs/input_for_mba.c -o input_for_mba
$LLVM_DIR/bin/opt -S -load-pass-plugin <build_dir>/lib/libMBAAdd.so -passes=mba-add input_
```

Note the `-debug-only=mba-add` and `-stats` flags in the command line - that's what enables the following output:

```
  %12 = add i8 %1, %0 ->   <badref> = add i8 111, %11
  %20 = add i8 %12, %2 ->   <badref> = add i8 111, %19
  %28 = add i8 %20, %3 ->   <badref> = add i8 111, %27
===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===

3 mba-add - The # of substituted instructions
```

As you can see, you get a nice summary from **MBAAdd**. In many cases this will be sufficient to understand what might be going wrong. Note that for these macros to work you need a debug build of LLVM (i.e. **opt**) and **llvm-tutor** (i.e. use `-DCMAKE_BUILD_TYPE=Debug` instead of `-DCMAKE_BUILD_TYPE=Release`).

For tricker issues just use a debugger. Below I demonstrate how to debug **MBAAdd**. More specifically, how to set up a breakpoint on entry to `MBAAdd::run`. Hopefully that will be sufficient for you to start.

Mac OS X

The default debugger on OS X is LLDB. You will normally use it like this:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
$LLVM_DIR/bin/clang -emit-llvm -S -O1 <source_dir>/inputs/input_for_mba.c -o input_for_mba
lldb -- $LLVM_DIR/bin/opt -S -load-pass-plugin <build_dir>/lib/libMBAAdd.dylib -passes=mba
(lldb) breakpoint set --name MBAAdd::run
(lldb) process launch
```

or, equivalently, by using LLDBs aliases:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
$LLVM_DIR/bin/clang -emit-llvm -S -O1 <source_dir>/inputs/input_for_mba.c -o input_for_mba
lldb -- $LLVM_DIR/bin/opt -S -load-pass-plugin <build_dir>/lib/libMBAAdd.dylib -passes=mba
(lldb) b MBAAdd::run
(lldb) r
```

At this point, LLDB should break at the entry to `MBAAdd::run`.

Ubuntu

On most Linux systems, GDB is the most popular debugger. A typical session will look like this:

```
export LLVM_DIR=<installation/dir/of/llvm/14>
$LLVM_DIR/bin/clang -emit-llvm -S -O1 <source_dir>/inputs/input_for_mba.c -o input_for_mba
gdb --args $LLVM_DIR/bin/opt -S -load-pass-plugin <build_dir>/lib/libMBAAdd.so -passes=mba
(gdb) b MBAAdd.cpp:MBAAdd::run
(gdb) r
```

At this point, GDB should break at the entry to `MBAAdd::run`.

About Pass Managers in LLVM

LLVM is a quite complex project (to put it mildly) and passes lay at its center - this is true for any multi-pass compiler. In order to manage the passes, a compiler needs a pass manager. LLVM currently enjoys not one, but two pass managers. This is important because depending on which pass manager you decide to use, the implementation of your pass (and in particular how you *register* it) will look slightly differently.

Overview of Pass Managers in LLVM

As I mentioned earlier, there are two pass managers in LLVM:

- *New Pass Manager* aka *Pass Manager* (that's how it is referred to in the code base), which is the default pass manager for the optimisation pipeline in LLVM.
- *Legacy Pass Manager*, which used to be the default pass managers for many years, is deprecated and will be removed after LLVM 14.

As the Legacy Pass Manager is to be deprecated in the near future (after LLVM 14), it is probably best to use the New Pass Manager for your passes. If you are already using it - that's great! Otherwise, don't worry, it's relatively straightforward to port passes from the Legacy to the New pass manager. In **llvm-tutor** all passes work with both pass managers.

New vs Legacy PM When Running Opt

We will demonstrate the difference between the pass managers when running **MBAAdd**. This is how you will use it with the legacy pass manager:

```
$LLVM_DIR/bin/opt -S --enable-new-pm=0 -load <build_dir>/lib/libMBAAdd.so -legacy-mba-add
```

And this is how you run it with the new pass manager:

```
$LLVM_DIR/bin/opt -S -load-pass-plugin <build_dir>/lib/libMBAAdd.so -passes=mba-add input_
```

There are three differences:

- the way you load your plugin: `-load` vs `-load-pass-plugin`
- the way you specify which pass/plugin to run: `-legacy-mba-add` vs `-passes=mba-add`
- if you want to use the Legacy Pass Manager, you need to use `--enable-new-pm=0` to disable the New Pass Manager

These differences stem from the fact that in the case of Legacy Pass Manager you register a new command line option for **opt**, whereas New Pass Manager simply requires you to define a pass pipeline (with `-passes=`).

Analysis vs Transformation Pass

The implementation of a pass depends on whether it is an Analysis or a Transformation pass. The difference in the API that you will use is often subtle and further differs between the pass managers.

For example, for the New Pass Manager:

- a transformation pass will normally inherit from PassInfoMixin,
- an analysis pass will inherit from AnalysisInfoMixin.

This is one of the key characteristics of the New Pass Managers - it makes the split into Analysis and Transformation passes very explicit. An Analysis pass requires a bit more bookkeeping and hence a bit more code. For example, you need to add an instance of
AnalysisKey
so that it can be identified by the New Pass Manager.

In the case of the Legacy Pass Manager, an Analysis pass is required to implement the print
method.
But otherwise, the API splits passes based on the unit of IR they operate on, e.g.
ModulePass
vs
FunctionPass.
This is one of the main differences between the pass managers in LLVM.

Note that for small standalone examples, the difference between Analysis and Transformation passes becomes less relevant.
HelloWorld is a good example. It does not
transform the input module, so in practice it is an Analysis pass. However, in

order to keep the implementation as simple as possible, I used the API for Transformation passes.

Within **llvm-tutor** the following passes can be used as reference Analysis and Transformation examples:

- **OpcodeCounter** - analysis pass
- **MBASub** - transformation pass

Other examples also adhere to LLVM's convention, but contain other complexities. Only in the case of **HelloWorld** simplicity was favoured over strictness.

Printing passes for the new pass manager

You might have noticed that some passes implement the `print` member method. This method is used by the Legacy Pass Manager to print the results of the corresponding Analysis pass. You can run it by passing the `-analyze` command line option when using **opt**. Interestingly, nothing of this sort is available in the New Pass Manager. Instead, you just implement a *printing pass*.

A printing pass for an Analysis pass is basically a Transformation pass that:

- requests the results of the analysis from the original pass
- prints these results.

In other words, it's just a wrapper pass. There's a convention to register such passes under the `print<analysis-pass-name>` command line option.

Dynamic vs Static Plugins

By default, all examples in **llvm-tutor** are built as dynamic plugins. However, LLVM provides infrastructure for both *dynamic* and *static* plugins (

documentation).
Static plugins are simply libraries linked into your executable (e.g. **opt**) statically. This way, unlike dynamic plugins, they don't require to be loaded at runtime with either `-load` or `-load-pass-plugin` options.

Static plugins are normally developed in-tree, i.e. within `llvm-project/llvm`, and all examples in **llvm-tutor** can be adapted to work this way. You can use static_registation.sh
to see it can be done for **MBASub**. This script will:

- copy the required source and test files into `llvm-project/llvm`
- adapt in-tree CMake scripts so that the in-tree version of **MBASub** is actually built
- remove `-load` and `-load-pass-plugin` from the in-tree tests for **MBASub**

Note that this script will modify `llvm-project/llvm`, but leave **llvm-tutor** intact. After running the script you will have to re-build **opt**. Two additional CMake flags have to be set: `LLVM_BUILD_EXAMPLES` and `LLVM_MBASUB_LINK_INTO_TOOLS`:

```
# LLVM_TUTOR_DIR: directory in which you cloned llvm-tutor
cd $LLVM_TUTOR_DIR
# LLVM_PROJECT_DIR: directory in which you cloned llvm-project
bash utils/static_registration.sh --llvm_project_dir $LLVM_PROJECT_DIR
# LLVM_BUILD_DIR: directory in which you previously built opt
cd $LLVM_BUILD_DIR
cmake -DLLVM_BUILD_EXAMPLES=On -DLLVM_MBASUB_LINK_INTO_TOOLS=On .
cmake --build . --target opt
```

Once **opt** is re-built, **MBASub** will be statically linked into **opt**. Now you can run it like this:

```
$LLVM_BUILD_DIR/bin/opt --passes=mba-sub -S $LLVM_TUTOR_DIR/test/MBA_sub.ll
```

Note that this time we didn't have to use `-load-pass-plugin` (or `-load`) to load **MBASub**. If you want to dive deeper into the required steps for static registration, you can scan `static_registation.sh` or run:

```
cd $LLVM_PROJECT_DIR
git diff
git status
```

This will print all the changes within `llvm-project/llvm` introduced by the script.

Optimisation Passes Inside LLVM

Apart from writing your own transformations an analyses, you may want to familiarize yourself with the passes available within LLVM. It is a great resource for learning how LLVM works and what makes it so powerful and successful. It is also a great resource for discovering how compilers work in general. Indeed, many of the passes implement general concepts known from the theory of compiler development.

The list of the available passes in LLVM can be a bit daunting. Below is a list of the selected few that are a good starting point. Each entry contains a link to the implementation in LLVM, a short description and a link to test files available within **llvm-tutor**. These test files contain a collection of annotated test cases for the corresponding pass. The goal of these tests is to demonstrate the functionality of the tested pass through relatively simple examples.

Name	Description	Test files in llvm-tutor
dce	Dead Code Elimination	dce.ll
memcpyopt	Optimise calls to `memcpy` (e.g. replace them with `memset`)	memcpyopt.ll
reassociate	Reassociate (e.g. 4 + (x + 5) -> x + (4 + 5)). This enables further optimisations, e.g. LICM.	reassociate.ll
always-	Always inlines functions decorated with `alwaysinline`	always-

Name	Description	Test files in llvm-tutor
inline		inline.ll
loop-deletion	Delete unused loops	loop-deletion.ll
licm	Loop-Invariant Code Motion (a.k.a. LICM)	licm.ll
slp	Superword-level parallelism vectorisation	slp_x86.ll, slp_aarch64.ll

This list focuses on LLVM's transform passes that are relatively easy to demonstrate through small, standalone examples. You can ran an individual test like this:

```
lit <source/dir/llvm/tutor>/test/llvm/always-inline.ll
```

To run an individual pass, extract one RUN line from the test file and run it:

```
$LLVM_DIR/bin/opt -inline-threshold=0 -always-inline -S <source/dir/llvm/tutor>/test/llvm/
```

References

Below is a list of LLVM resources available outside the official online documentation that I have found very helpful. Where possible, the items are sorted by date.

- **LLVM IR**
  - *"LLVM IR Tutorial-Phis,GEPs and other things, ohmy!"*, V.Bridgers, F. Piovezan, EuroLLVM, (slides, video)
  - "*Mapping High Level Constructs to LLVM IR*", M. Rodler (link)

- **Examples in LLVM**
    - Control Flow Graph simplifications: llvm/examples/IRTransforms/
    - Hello World Pass: llvm/lib/Transforms/Hello/
    - Good Bye World Pass: llvm/examples/Bye/
- **LLVM Pass Development**
    - "*Writing an LLVM Optimization*", Jonathan Smith video
    - "*Getting Started With LLVM: Basics* ", J. Paquette, F. Hahn, LLVM Dev Meeting 2019 video
    - "*Writing an LLVM Pass: 101*", A. Warzyński, LLVM Dev Meeting 2019 video
    - "*Writing LLVM Pass in 2018*", Min-Yih Hsu blog
    - "*Building, Testing and Debugging a Simple out-of-tree LLVM Pass*" Serge Guelton, Adrien Guinet, LLVM Dev Meeting 2015 (slides, video)
- **Legacy vs New Pass Manager**
    - "*New PM: taming a custom pipeline of Falcon JIT*", F. Sergeev, EuroLLVM 2018 (slides, video)
    - "*The LLVM Pass Manager Part 2*", Ch. Carruth, LLVM Dev Meeting 2014 (slides, video)
    - *"Passes in LLVM, Part 1"*, Ch. Carruth, EuroLLVM 2014 (slides, video)
- **LLVM Based Tools Development**
    - "*Introduction to LLVM*", M. Shah, Fosdem 2018, link
    - "*Building an LLVM-based tool. Lessons learned*", A. Denisov, blog, video

Credits

This is first and foremost a community effort. This project wouldn't be

possible without the amazing LLVM online documentation, the plethora of great comments in the source code, and the llvm-dev mailing list. Thank you!

It goes without saying that there's plenty of great presentations on YouTube, blog posts and GitHub projects that cover similar subjects. I've learnt a great deal from them - thank you all for sharing! There's one presentation/tutorial that has been particularly important in my journey as an aspiring LLVM developer and that helped to *democratise* out-of-source pass development:

- "Building, Testing and Debugging a Simple out-of-tree LLVM Pass" Serge Guelton, Adrien Guinet (slides, video)

Adrien and Serge came up with some great, illustrative and self-contained examples that are great for learning and tutoring LLVM pass development. You'll notice that there are similar transformation and analysis passes available in this project. The implementations available here reflect what **I** found most challenging while studying them.

License