Tutorial for building tools using LibTooling and LibASTMatchers

This document is intended to show how to build a useful source-to-source translation tool based on Clang's **LibTooling**. It is explicitly aimed at people who are new to Clang, so all you should need is a working knowledge of C++ and the command line.

In order to work on the compiler, you need some basic knowledge of the abstract syntax tree (AST). To this end, the reader is incouraged to skim the **Introduction to the Clang AST**

Step 0: Obtaining Clang

As Clang is part of the LLVM project, you'll need to download LLVM's source code first. Both Clang and LLVM are maintained as Subversion repositories, but we'll be accessing them through the git mirror. For further information, see the **getting started guide**.

```
mkdir ~/clang-llvm && cd ~/clang-llvm
git clone http://llvm.org/git/llvm.git
cd llvm/tools
git clone http://llvm.org/git/clang.git
cd clang/tools
git clone http://llvm.org/git/clang-tools-extra.git extra
```

Next you need to obtain the CMake build system and Ninja build tool. You may already have CMake installed, but current binary versions of CMake aren't built with Ninja support.

```
cd ~/clang-llvm
git clone https://github.com/martine/ninja.git
cd ninja
git checkout release
./bootstrap.py
sudo cp ninja /usr/bin/

cd ~/clang-llvm
git clone git://cmake.org/stage/cmake.git
cd cmake
git checkout next
./bootstrap
make
sudo make install
```

Okay. Now we'll build Clang!

```
cd ~/clang-llvm
mkdir build && cd build
cmake -G Ninja ../llvm -DLLVM_BUILD_TESTS=ON  # Enable tests; default is off.
ninja
ninja check       # Test LLVM only.
ninja clang-test  # Test Clang only.
ninja install
```

And we're live.

All of the tests should pass, though there is a (very) small chance that you can catch LLVM and Clang out of sync. Running `'git svn rebase'` in both the llvm and clang directories should fix any problems.

Finally, we want to set Clang as its own compiler.

```
cd ~/clang-llvm/build
ccmake ../llvm
```

The second command will bring up a GUI for configuring Clang. You need to set the entry for `CMAKE_CXX_COMPILER`. Press `'t'` to turn on advanced mode. Scroll down to `CMAKE_CXX_COMPILER`, and set it to `/usr/bin/clang++`, or wherever you installed it. Press `'c'` to configure, then `'g'` to generate CMake's files.

Finally, run ninja one last time, and you're done.

Step 1: Create a ClangTool

Now that we have enough background knowledge, it's time to create the simplest productive ClangTool in existence: a syntax checker. While this already exists as `clang-check`, it's important to understand what's going on.

First, we'll need to create a new directory for our tool and tell CMake that it exists. As this is not going to be a core clang tool, it will live in the `tools/extra` repository.

```
cd ~/clang-llvm/llvm/tools/clang
mkdir tools/extra/loop-convert
echo 'add_subdirectory(loop-convert)' >> tools/extra/CMakeLists.txt
vim tools/extra/loop-convert/CMakeLists.txt
```

CMakeLists.txt should have the following contents:

```
set(LLVM_LINK_COMPONENTS support)

add_clang_executable(loop-convert
  LoopConvert.cpp
  )
target_link_libraries(loop-convert
  clangTooling
  clangBasic
  clangASTMatchers
  )
```

With that done, Ninja will be able to compile our tool. Let's give it something to compile! Put the following into `tools/extra/loop-convert/LoopConvert.cpp`. A detailed explanation of why the different parts are needed can be found in the **LibTooling documentation**.

```
// Declares clang::SyntaxOnlyAction.
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
// Declares llvm::cl::extrahelp.
#include "llvm/Support/CommandLine.h"

using namespace clang::tooling;
using namespace llvm;

// Apply a custom category to all command-line options so that they are the
// only ones displayed.
static llvm::cl::OptionCategory MyToolCategory("my-tool options");

// CommonOptionsParser declares HelpMessage with a description of the common
// command-line options related to the compilation database and input files.
// It's nice to have this help message in all tools.
static cl::extrahelp CommonHelp(CommonOptionsParser::HelpMessage);

// A help message for this specific tool can be added afterwards.
static cl::extrahelp MoreHelp("\nMore help text...");

int main(int argc, const char **argv) {
  CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);
  ClangTool Tool(OptionsParser.getCompilations(),
                 OptionsParser.getSourcePathList());
  return Tool.run(newFrontendActionFactory<clang::SyntaxOnlyAction>().get());
}
```

And that's it! You can compile our new tool by running ninja from the `build` directory.

```
cd ~/clang-llvm/build
ninja
```

You should now be able to run the syntax checker, which is located in `~/clang-llvm/build/bin`, on any source file. Try it!

```
echo "int main() { return 0; }" > test.cpp
bin/loop-convert test.cpp --
```

Note the two dashes after we specify the source file. The additional options for the compiler are passed after the dashes rather than loading them from a compilation database - there just aren't any options needed right now.

Intermezzo: Learn AST matcher basics

Clang recently introduced the **ASTMatcher library** to provide a simple, powerful, and concise way to describe specific patterns in the AST. Implemented as a DSL powered by macros and templates (see **ASTMatchers.h** if you're curious), matchers offer the feel of algebraic data types common to functional programming languages.

For example, suppose you wanted to examine only binary operators. There is a matcher to do exactly that, conveniently named `binaryOperator`. I'll give you one guess what this matcher does:

```
binaryOperator(hasOperatorName("+"), hasLHS(integerLiteral(equals(0))))
```

Shockingly, it will match against addition expressions whose left hand side is exactly the literal 0. It will not match against other forms of 0, such as `'\0'` or `NULL`, but it will match against macros that expand to 0. The matcher will also not match against calls to the overloaded operator `'+'`, as there is a separate `operatorCallExpr` matcher to handle overloaded operators.

There are AST matchers to match all the different nodes of the AST, narrowing matchers to only match AST nodes fulfilling specific criteria, and traversal matchers to get from one kind of AST node to another. For a complete list of AST matchers, take a look at the **AST Matcher References**

All matcher that are nouns describe entities in the AST and can be bound, so that they can be referred to whenever a match is found. To do so, simply call the method `bind` on these matchers, e.g.:

```
variable(hasType(isInteger())).bind("intvar")
```

Step 2: Using AST matchers

Okay, on to using matchers for real. Let's start by defining a matcher which will capture all `for` statements that define a new variable initialized to zero. Let's start with matching all `for` loops:

```
forStmt()
```

Next, we want to specify that a single variable is declared in the first portion of the loop, so we can extend the matcher to

```
forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl()))))
```

Finally, we can add the condition that the variable is initialized to zero.

```
forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl(
  hasInitializer(integerLiteral(equals(0))))))))
```

It is fairly easy to read and understand the matcher definition ("match loops whose init portion declares a single variable which is initialized to the integer literal 0"), but deciding that every piece is necessary is more difficult. Note that this matcher will not match loops whose variables are initialized to `'\0'`, `0.0`, `NULL`, or any form of zero besides the integer 0.

The last step is giving the matcher a name and binding the `ForStmt` as we will want to do something with it:

```
StatementMatcher LoopMatcher =
  forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl(
    hasInitializer(integerLiteral(equals(0)))))))).bind("forLoop");
```

Once you have defined your matchers, you will need to add a little more scaffolding in order to run them. Matchers are paired with a `MatchCallback` and registered with a `MatchFinder` object, then run from a `ClangTool`. More code!

Add the following to `LoopConvert.cpp`:

```
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"

using namespace clang;
using namespace clang::ast_matchers;

StatementMatcher LoopMatcher =
  forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl(
    hasInitializer(integerLiteral(equals(0)))))))).bind("forLoop");

class LoopPrinter : public MatchFinder::MatchCallback {
public :
  virtual void run(const MatchFinder::MatchResult &Result) {
    if (const ForStmt *FS = Result.Nodes.getNodeAs<clang::ForStmt>("forLoop"))
      FS->dump();
  }
};
```

And change `main()` to:

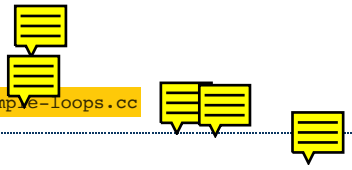
```
int main(int argc, const char **argv) {
  CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);
  ClangTool Tool(OptionsParser.getCompilations(),
                 OptionsParser.getSourcePathList());

  LoopPrinter Printer;
  MatchFinder Finder;
  Finder.addMatcher(LoopMatcher, &Printer);

  return Tool.run(newFrontendActionFactory(&Finder).get());
}
```

Now, you should be able to recompile and run the code to discover for loops. Create a new file with a few examples, and test out our new handiwork:

```
cd ~/clang-llvm/llvm/llvm_build/
ninja loop-convert
vim ~/test-files/simple-loops.cc
bin/loop-convert ~/test-files/simple-loops.cc
```



Step 3.5: More Complicated Matchers

Our simple matcher is capable of discovering for loops, but we would still need to filter out many more ourselves. We can do a good portion of the remaining work with some cleverly chosen matchers, but first we need to decide exactly which properties we want to allow.

How can we characterize for loops over arrays which would be eligible for translation to range-based syntax? Range based loops over arrays of size `N` that:

- start at index `0`
- iterate consecutively
- end at index `N-1`

We already check for (1), so all we need to add is a check to the loop's condition to ensure that the loop's index variable is compared against `N` and another check to ensure that the increment step just increments this same variable. The matcher for (2) is straightforward: require a pre- or post-increment of the same variable declared in the init portion.

Unfortunately, such a matcher is impossible to write. Matchers contain no logic for comparing two arbitrary AST nodes and determining whether or not they are equal, so the best we can do is matching more than we would like to allow, and punting extra comparisons to the callback.

In any case, we can start building this sub-matcher. We can require that the increment step be a unary increment like this:

```
hasIncrement(unaryOperator(hasOperatorName("++")))
```

Specifying what is incremented introduces another quirk of Clang's AST: Usages of variables are represented as `DeclRefExpr`'s ("declaration reference expressions") because they are expressions which refer to variable declarations. To find a `unaryOperator` that refers to a specific

declaration, we can simply add a second condition to it:

```
hasIncrement(unaryOperator(
  hasOperatorName("++"),
  hasUnaryOperand(declRefExpr())))
```

Furthermore, we can restrict our matcher to only match if the incremented variable is an integer:

```
hasIncrement(unaryOperator(
  hasOperatorName("++"),
  hasUnaryOperand(declRefExpr(to(varDecl(hasType(isInteger())))))))
```

And the last step will be to attach an identifier to this variable, so that we can retrieve it in the callback:

```
hasIncrement(unaryOperator(
  hasOperatorName("++"),
  hasUnaryOperand(declRefExpr(to(
    varDecl(hasType(isInteger())).bind("incrementVariable"))))))
```

We can add this code to the definition of `LoopMatcher` and make sure that our program, outfitted with the new matcher, only prints out loops that declare a single variable initialized to zero and have an increment step consisting of a unary increment of some variable.

Now, we just need to add a matcher to check if the condition part of the `for` loop compares a variable against the size of the array. There is only one problem - we don't know which array we're iterating over without looking at the body of the loop! We are again restricted to approximating the result we want with matchers, filling in the details in the callback. So we start with:

```
hasCondition(binaryOperator(hasOperatorName("<"))
```

It makes sense to ensure that the left-hand side is a reference to a variable, and that the right-hand side has integer type.

```
hasCondition(binaryOperator(
  hasOperatorName("<"),
  hasLHS(declRefExpr(to(varDecl(hasType(isInteger()))))),
  hasRHS(expr(hasType(isInteger())))))
```

Why? Because it doesn't work. Of the three loops provided in `test-files/simple.cpp`, zero of them have a matching condition. A quick look at the AST dump of the first for loop, produced by the previous iteration of loop-convert, shows us the answer:

```
(ForStmt 0x173b240
  (DeclStmt 0x173afc8
    0x173af50 "int i =
      (IntegerLiteral 0x173afa8 'int' 0)")
  <<<>>>
  (BinaryOperator 0x173b060 '_Bool' '<'
    (ImplicitCastExpr 0x173b030 'int'
      (DeclRefExpr 0x173afe0 'int' lvalue Var 0x173af50 'i' 'int'))
    (ImplicitCastExpr 0x173b048 'int'
      (DeclRefExpr 0x173b008 'const int' lvalue Var 0x170fa80 'N' 'const int')))
  (UnaryOperator 0x173b0b0 'int' lvalue prefix '++'
    (DeclRefExpr 0x173b088 'int' lvalue Var 0x173af50 'i' 'int'))
  (CompoundStatement ...
```

We already know that the declaration and increments both match, or this loop wouldn't have been dumped. The culprit lies in the implicit cast applied to the first operand (i.e. the LHS) of the less-than operator, an L-value to R-value conversion applied to the expression referencing `i`. Thankfully, the matcher library offers a solution to this problem in the form of `ignoringParenImpCasts`, which instructs the matcher to ignore implicit casts and parentheses before continuing to match. Adjusting the condition operator will restore the desired match.

```
hasCondition(binaryOperator(
  hasOperatorName("<"),
  hasLHS(ignoringParenImpCasts(declRefExpr(
    to(varDecl(hasType(isInteger())))))),
  hasRHS(expr(hasType(isInteger())))))
```

After adding binds to the expressions we wished to capture and extracting the identifier strings into variables, we have array-step-2 completed.

Step 4: Retrieving Matched Nodes

So far, the matcher callback isn't very interesting: it just dumps the loop's AST. At some point, we will need to make changes to the input source code. Next, we'll work on using the nodes we bound in the previous step.

The `MatchFinder::run()` callback takes a `MatchFinder::MatchResult&` as its parameter. We're most interested in its `Context` and `Nodes` members. Clang uses the `ASTContext` class to represent contextual information about the AST, as the name implies, though the most functionally important detail is that several operations require an `ASTContext*` parameter. More immediately useful is the set of matched nodes, and how we retrieve them.

Since we bind three variables (identified by ConditionVarName, InitVarName, and IncrementVarName), we can obtain the matched nodes by using the `getNodeAs()` member function.

In `LoopConvert.cpp` add

```
#include "clang/AST/ASTContext.h"
```

Change `LoopMatcher` to

```
StatementMatcher LoopMatcher =
    forStmt(hasLoopInit(declStmt(
                hasSingleDecl(varDecl(hasInitializer(integerLiteral(equals(0))))
                              .bind("initVarName")))),
            hasIncrement(unaryOperator(
                hasOperatorName("++"),
                hasUnaryOperand(declRefExpr(
                    to(varDecl(hasType(isInteger())).bind("incVarName")))))),
            hasCondition(binaryOperator(
                hasOperatorName("<"),
                hasLHS(ignoringParenImpCasts(declRefExpr(
                    to(varDecl(hasType(isInteger())).bind("condVarName"))))),
                hasRHS(expr(hasType(isInteger()))))).bind("forLoop");
```

And change `LoopPrinter::run` to

```
void LoopPrinter::run(const MatchFinder::MatchResult &Result) {
  ASTContext *Context = Result.Context;
  const ForStmt *FS = Result.Nodes.getNodeAs<ForStmt>("forLoop");
  // We do not want to convert header files!
  if (!FS || !Context->getSourceManager().isWrittenInMainFile(FS->getForLoc()))
    return;
  const VarDecl *IncVar = Result.Nodes.getNodeAs<VarDecl>("incVarName");
  const VarDecl *CondVar = Result.Nodes.getNodeAs<VarDecl>("condVarName");
  const VarDecl *InitVar = Result.Nodes.getNodeAs<VarDecl>("initVarName");

  if (!areSameVariable(IncVar, CondVar) || !areSameVariable(IncVar, InitVar))
    return;
  llvm::outs() << "Potential array-based loop discovered.\n";
}
```

Clang associates a `VarDecl` with each variable to represent the variable's declaration. Since the "canonical" form of each declaration is unique by address, all we need to do is make sure neither `ValueDecl` (base class of `VarDecl`) is `NULL` and compare the canonical Decls.

```
static bool areSameVariable(const ValueDecl *First, const ValueDecl *Second) {
  return First && Second &&
         First->getCanonicalDecl() == Second->getCanonicalDecl();
}
```

If execution reaches the end of `LoopPrinter::run()`, we know that the loop shell that looks like

```
for (int i= 0; i < expr(); ++i) { ... }
```

For now, we will just print a message explaining that we found a loop. The next section will deal with recursively traversing the AST to discover all changes needed.

As a side note, it's not as trivial to test if two expressions are the same, though Clang has already done the hard work for us by providing a way to canonicalize expressions:

```
static bool areSameExpr(ASTContext *Context, const Expr *First,
                        const Expr *Second) {
  if (!First || !Second)
    return false;
  llvm::FoldingSetNodeID FirstID, SecondID;
  First->Profile(FirstID, *Context, true);
  Second->Profile(SecondID, *Context, true);
  return FirstID == SecondID;
}
```

This code relies on the comparison between two `llvm::FoldingSetNodeIDs`. As the documentation for `Stmt::Profile()` indicates, the `Profile()` member function builds a description of a node in the AST, based on its properties, along with those of its children. `FoldingSetNodeID` then serves as a hash we can use to compare expressions. We will need `areSameExpr` later. Before you run the new code on the additional loops added to test-files/simple.cpp, try to figure out which ones will be considered potentially convertible.