Improving code reuse in clang tools with clangmetatool

Daniel Ruoso
druoso@bloomberg.net
Bloomberg

October 17, 2018

Static Analysis and Automated Refactoring at Bloomberg

-
-
-

Static Analysis and Automated Refactoring at Bloomberg

- 30+ years of code
-
-

Static Analysis and Automated Refactoring at Bloomberg

- 30+ years of code
- substantial amount of reuse
-

Static Analysis and Automated Refactoring at Bloomberg

- 30+ years of code
- substantial amount of reuse
- continuously integrated and deployed

Writing Language Tools – A brief History

Writing Language Tools – A brief History

- tools space with gcc
-
-

Writing Language Tools – A brief History

- tools space with gcc
- llvm3.8 boom
-

Writing Language Tools – A brief History

- tools space with gcc
- llvm3.8 boom
- clangTooling

My first clang tool

-
-
-

My first clang tool

- exercise: re-implement include-what-you-use

My first clang tool

- exercise: re-implement include-what-you-use
- unsure about life-cycle? just use globals
-

My first clang tool

- exercise: re-implement include-what-you-use
- unsure about life-cycle? just use globals
- unsure about when to rewrite? just rewrite asap

My first clang tool

-
-
-

My first clang tool

- so many stub doxygen docs
-
-

My first clang tool

- so many stub doxygen docs
- so many callbacks
-

My first clang tool

- so many stub doxygen docs
- so many callbacks
- life-cycle of objects unclear

My first clang tool – Lessons

-
-
-

My first clang tool – Lessons

- writing a clang tool is actually not that hard

My first clang tool – Lessons

- writing a clang tool is actually not that hard
- not a single line of reusable code
-

My first clang tool – Lessons

- writing a clang tool is actually not that hard
- not a single line of reusable code
- tightly coupling: analysis, rewriting, data collection

Principles

Principles

- Refactoring tool should make smallest possible change

Principles

- Refactoring tool should make smallest possible change
- Create the tool, run it, throw it away
-

Principles

- Refactoring tool should make smallest possible change
- Create the tool, run it, throw it away
- Design Patterns: Collect, Analyze, Rewrite

Design Pattern: Data Collectors

Design Pattern: Data Collectors

- Register callbacks, stores data in member
-
-

Design Pattern: Data Collectors

- Register callbacks, stores data in member
- No specific analysis performed
-

Design Pattern: Data Collectors

- Register callbacks, stores data in member
- No specific analysis performed
- Expose the data in a useful way

Design Pattern: Analysis

►

►

►

Design Pattern: Analysis

- Single entry point
-
-

Design Pattern: Analysis

- Single entry point
- Straight-forward imperative code
-

Design Pattern: Analysis

- Single entry point
- Straight-forward imperative code
- As little tool-specific code as possible

Design Pattern: Refactoring

-
-
-

Design Pattern: Refactoring

- Already part of the tooling API
-
-

Design Pattern: Refactoring

- Already part of the tooling API
- Just fill in the ReplacementsMap
-

Design Pattern: Refactoring

- Already part of the tooling API
- Just fill in the ReplacementsMap
- Handles coherency for you

clangmetatool

- Life-cycle management
- Data collectors
- Reusable Analysis

clangmetatool: life-cycle management

```
int main(int argc, const char* argv[]) {
  llvm::cl::OptionCategory MyToolCategory("my-tool options");
  llvm::cl::extrahelp CommonHelp
    (clang::tooling::CommonOptionsParser::HelpMessage);
  clang::tooling::CommonOptionsParser
    optionsParser(argc, argv, MyToolCategory);
  clang::tooling::RefactoringTool tool(optionsParser.getCompilations(),
                                       optionsParser.getSourcePathList());
  clangmetatool::MetaToolFactory< clangmetatool::MetaTool<MyTool> >
    raf(tool.getReplacements());
  int r = tool.runAndSave(&raf);
  return r;
}
```

clangmetatool: life-cycle management

```
class MyTool {
private:
  SomeDataCollector collector1;
  SomeOtherDataCollector collector2;
public:
  MyTool(clang::CompilerInstance* ci, clang::ast_matchers::MatchFinder *f)
   :collector1(ci, f), collector2(ci, f) {
   // the individual collectors will register their callbacks in their
   // constructor, the tool doesn't really need to do anything else here.
  }
  void postProcessing
  (std::map<std::string, clang::tooling::Replacements> &replacementsMap) {
   // use data from collector1 and collector2
   // generate warnings and notices
   // add replacements to replacementsMap
  }
};
```

clangmetatool: reusable data-collector

```
class WhoCallsIt {
private:
  clangmetatool::collectors::FindCalls fc;
public:
  MyTool(clang::CompilerInstance* ci, clang::ast_matchers::MatchFinder *f)
   :(ci, f, "legacyfunction") {
  }
  void postProcessing
  (std::map<std::string, clang::tooling::Replacements> &replacementsMap) {
    FindCallsData *fcd = fc.getData();
    auto calls_it = fcd->call_ref.begin();
    while (calls_it != fcd->call_ref.end()) {
      // do something for each call to legacyfunction
    }
  }
};
```

clangmetatool: reusable analysis

```
clangmetatool::propagation::ConstantCStringPropagator prop(ci);
PropagationResult<std::string> r = prop.runPropagation(funcdecl, vdrefexpr);
if (!r.isUnresolved()) {
  std::cout
  << "value of variable at this point is "
  << r.getResult()
  << std::endl;
}
```

Impact at Bloomberg

- low cost to writing new tools
- custom static analysis accessible
- automated refactoring on the rise

Questions?

druoso@bloomberg.net

`https://bloomberg.github.io/clangmetatool`