Writing Your Own clang-tidy Rules

Simon Griebel
Munich LLVM Meetup 2024-04-29
Motivation

- Personal Motivation: I am interested in applying more formal verification methods (i.e. static analysis)
  - Not so much interest in automatic refactoring (fix rules)
- The first opportunity to push this in the team: A crash due to an exception escaping from a noexcept function.
  - In theory easy to verify
  - But happens very rarely in practice and the final check is problematic due to false positives (not totally unexpected)
Motivation – Can you spot the bug?

A much better use case for a clang-tidy rule:

```cpp
int64_t simpleSum(const std::vector<int64_t>& vec) {
    return std::accumulate(std::begin(vec), std::end(vec), 0, [](int64_t acc, int64_t val){
        return acc + val;
    });
}
```

Neither clang (with -Weverything) nor gcc (with all relevant warnings I know) find this.
Motivation – Can you spot the bug?

The accumulator has type int (not int64_t), which means that the accumulate actually returns an int.

```cpp
int64_t simpleSum(const std::vector<int64_t>& vec) {
    return std::accumulate(std::begin(vec), std::end(vec), 0, [](int64_t acc, int64_t val){
        return acc + val;
    });
}
```

The compilers do not flag this because the actual code is in a system header.
How to get started – Option 1 – Fork clang-tidy

- **Advantages:**
  - There is a script that generates the boilerplate for new rules.
  - You get direct access to the testing framework
  - You can reuse any runner scripts

- **Disadvantages:**
  - You have to maintain a full fork of llvm
  - You always have to build llvm (can be cached, but still not very nice for sporadic contributors)
How to get started – Option 2 – Write your own “clang-tidy”

- The advantages/disadvantages are essentially the reverse of the previous slide (no framework, smaller codebase to maintain)
- The code that you have to reimplement for a simple version is not that complicated. The missing runner scripts are probably the most annoying point.
- Potential to build a static-analysis tool that is more powerful than clang-tidy.
How to get started – Option 3 – Write a clang-tidy plugins

- Build a shared library that links clang-tidy in which you write the rules as normal clang-tidy rules and add a bit of boilerplate.
- Essentially combines the advantages of the previous two: you can reuse the framework and only have to maintain a small project.
- You only miss the clang-tidy testing framework. I would assume that this is not too hard to copy.
- Links to example codebases: [1](#) [2](#)
- If you write new rules just use a plugin.
Before we get started: A bit about the clang AST

- **Decl**
  - CxxRecordDecl (= struct/class declaration)
  - **FunctionDecl**
    - CXXConstructorDecl
  - VarDecl
  - ...

- **Stmt**
  - Expr
    - CallExpr
    - CXXConstructExpr
  - DeclStmt
  - CxxTryStmt
  - ...

A constructor declaration is a function declaration, but a constructor call is not a function call.

The indentation matches the inheritance hierarchy.

A *Type* can be seen as an annotation of statements/declarations.
Before we get started: A bit about the clang AST

There is (at least) one more relevant top level AST node type:

- Decl
  - ...
- Stmt
  - ...
- ...


Before we get started: A bit about the clang AST

There is (at least) one more relevant top level AST node type:

- Decl
  - …
- Stmt
  - …
- CXXCtorInitializer

```cpp
class X{
  public:
  X (int x) : x_(x) {}  // Highlighted line
  
  private:
  int x_;  
}
```
Printing the AST

- You can generate a full ast dump by adding the "-Xclang -ast-dump" compile options
- You can use clang-query (example output below)

```
clang-query> enable output dump
clang-query> m functionDecl()

Match #1:

  1 | int f(){
  2 | ^~~~~~~~~
  3 | return 3;
  4 | ~~~~~~~~~
  5 |

Binding for "root":
  FunctionDecl 0x5e0ab9dd70a0 <home/simon/cpp_tests/clang-query-test.cpp:1:1, line:3:1> line:1:5 used f 'int ()'
    -CompoundStmt 0x5e0ab9dd71c0 <col:8, line:3:1>
      -ReturnStmt 0x5e0ab9dd71b0 <line:2:3, col:10>
        -IntegerLiteral 0x5e0ab9dd7190 <col:10> 'int' 3
```
Let’s write a check – Disclaimers

- I had help from Siqi Ling and Lucas Wolf
- The rules are not peer reviewed so they probably not optimal and might contain bugs.
- I am not an expert in the clang-AST. I got my knowledge from experimenting with clang-tidy.
Let’s write a check – The basic structure

A rule is split into two parts:

● The matcher
  ○ Matches AST-Nodes
  ○ Uses a special syntax.
  ○ They are powerful, but have their limits.

● The check
  ○ Contains more complicated logic (on the previously matched nodes)
  ○ A C++ function that can inspect the AST
  ○ Contains the generation of the diagnostic (and also of the automatic fix if there is one)
A matcher that finds all std::accumulate calls

callExpr(hasDeclaration(functionDecl(hasName("::std::accumulate")))).bind("f");

The bind is used to reference (parts of) matched AST-nodes in the check.

The [AST Matcher Reference](#). If you click on the matchers, you get even more info.
invocation(
    hasDeclaration(functionDecl(unless(isNoThrow()))),
    hasAncestor(functionDecl(isNoThrow()).bind("caller")),
    optionally(hasAncestor(cxxTryStmt().bind("try")))
).bind("call");

What it should do: Find all functions or constructor calls that are not noexcept and that are part of a noexcept function (or constructor), the “caller”. If the call is inside of a try-statement, bind the latter to “try”.

We do not check throw statements, because there is already a clang-tidy rule for that.
The matcher for the noexcept check – Buggy version

invocation(
    hasDeclaration(functionDecl(unless(isNoThrow()))),
    hasAncestor(functionDecl(isNoThrow()).bind("caller")),
    optionally(hasAncestor(cxxTryStmt().bind("try"))))
  .bind("call");

*hasAncestor* relates to the full AST and **makes no guarantee that the matched statement is in the same declaration context.**
The matcher for the noexcept check - Buggy version

An example for a match with the rule:

Match #1:

```
  4 | int f() noexcept {
        ^~~~~~~~~~~~~~
  5 |   auto lambda{}[] { canThrow(); };
      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  6 |   return 3;
      ~~~~~~~
  7 | }
     ~

  5 | auto lambda{}[] { canThrow(); };
        ^~~~~~~~
```
The matcher for the noexcept check – Better version

invocation(
  hasDeclaration(functionDecl(unless(isNoThrow()))),
  hasAncestor(functionDecl().bind("caller")),
  optionally(hasAncestor(cxxTryStmt(hasAncestor(functionDecl(equalsBoundNode("caller"))))
    .bind("try"))
  ).bind("call");

We now match the first ancestor function declaration (=the function declaration the invocation is actually in), the “caller”, and the we only match the “closest” try-statement, if it is directly in the “caller”.
The C++ part for the noexcept check – The Basics

Ignoring the the try-block handling for now.

```c++
void TransitiveNoexceptCheck::check(const MatchFinder::MatchResult& Result) {
    auto invocationExpr{Result.Nodes.getNodeAs<clang::Expr>("call")};
    auto callerDecl{Result.Nodes.getNodeAs<clang::FunctionDecl>("caller")};

    if (!isNothrow(callerDecl)) {
        return;
    }

    diag(invocationExpr->getLocation(), "Call to non-noexcept function from noexcept context");
    diag(callerDecl->getLocation(), "noexcept function declared here", clang::DiagnosticIDs::Note);
}
```

Additional diagnostic lines always have to be notes!
For isNothrow I copied the implementation of the corresponding matcher.
A quite hacky solution that checks whether the try has a catch(...) block.

```cpp
auto const* const EnclosingTry = Result.Notes.getNodeAs<clang::CXXTryStmt>("try");
if (EnclosingTry != nullptr) {
    for (size_t I = 0; I < EnclosingTry->getNumHandlers(); ++I) {
        if (auto* handler = EnclosingTry->getHandler(I);
            handler->getExceptionDecl() == nullptr) {
            const StatementMatcher ThrowMatcher = forEachDescendant(cxxThrowExpr());
            break;
        }
    }
}
```

The match has again the “declaration context” problem, but here the chance for a false positive should be quite low.
The accumulate check – First try

- In first matcher we matched all accumulate calls.
- At first **we looked at the types of the accumulator, the lambda and the iterators.**
- **As it turns out especially the latter is quite hard to do.**
  - You can use the return type of operator* or the type of the “value_type” typedef.
  - **Issues:**
    - You need to find it even if it is defined in a parent class (where I do not know whether you can do it nicely)
    - You need to handle the case, when the iterators are just pointers and do not have a CXXRecordDecl.
- **This does not scale well for functions with slightly different parameters** (like `inner_product`, which has two lambda function parameters)
The accumulate check – Second try

- Taking a step back: What we wanted to do is to check whether there is a narrowing conversion in the template instantiation.
- Let’s do just that.
The accumulate check – Second try – The matcher

callExpr(hasDeclaration(functionDecl(
    hasAnyName("::std::accumulate", "::std::iota"),
    hasBody(stmt(forEachDescendant(
        castExpr(anyOf(hasCastKind(clang::CK___IntegralCast),...),
        hasType(qualType().bind("target-type")),
        hasSourceExpression(expr(hasType(qualType().bind("source-type")))))
            .bind("cast-expr")))))
    .bind("call-expr");

For all accumulate and std::iota calls, look for all castExpressions (one by one) in their function bodies of the given kinds and bind their result type to “target-type” and the result type of its source expression to source-type.

The forEachDescendant call has again the declaration context issue (but here it should not matter that much).
The accumulate check – Second try – The check

Essentially, the only things left to do are:

- **Get the canonical types** without qualifiers and references to be able to compare the types
- Do the narrowing check (code mostly copied from clang)
- Print the diagnostics
Preliminary Results

● The accumulate rule seems to be working fine and I will try to polish it and see if I can push it upstream.

● The noexcept rule seems to be too noisy to be really useful.
Conclusion

Writing an initial clang-tidy rule is often quite easy. Really making it watertight is very hard, because you run into a lot of edge cases in the AST, which just was not built for verification. Often the initial rule already helps.
Future Work

- It would be nice to do analysis over the call-graph.
- For that it would be nice to easily transfer analysis state between compilation units.
- Does anybody know what the current state of running multiple clang instances in the same process is?
Thank you.