FleCSI Static Analysis

13 August 2018

Martin Staley

CCS-7: Applied Computer Science

Introduction

In order to help ensure FleCSI's appeal to users and potential users, we'd like to prevent or clarify, at *compile time* or earlier, certain problems that would otherwise not arise until run-time, or which would be unclear to users.

For example:

- A FleCSI code might compile but not run, due to an issue that could actually be detected early. Example: **executing a task that wasn't registered**, either because registration was simply forgotten, or because a name – to be stringified and hashed – was misspelled somewhere.

- Other mistakes may trigger errors that mask the real problem. Example: sending the `flecsi_register_data_client()` macro a **`client_type` that isn't derived from `data::data_client_t`**.

Introduction (continued)

- Macro use (e.g. for the task registrations/executions) is presently a quasi-necessary evil for FleCSI users. **Macros obfuscate code** and compound our problems. (We say quasi-necessary because a user could always write, directly, what a macro emits, but I believe we're discouraging that. And, doing so won't generally prevent the errors we'll speak of.)

- C++ compilers often emit **cryptic error messages**. (For general C++ codes, of course; not just for FleCSI!) Can we improve this situation, at least in the context of certain error messages relating to FleCSI codes?

- Bear in mind that **users generally aren't computer scientists**. As such, they're probably less equipped than we are to see the clear error tree behind the cryptic, convoluted message forest.

FleCSI Analyzer

Our goal, then, is to provide a tool that performs additional “compile time” (or before-compile-time) analysis of FleCSI codes. We could have written a standalone analyzer, but chose instead to integrate our analysis into `Clang`. In particular, we’re branching from Kitsune Clang:

`https://github.com/lanl/kitsune`

Relative to a standalone analyzer, Clang provides several advantages. Obvious ones are proper **lexing and parsing** of all C++ code. Less obvious benefits stem from the **code analysis** Clang necessarily performs. We can, for instance, easily query whether one class derives from another.

Users will therefore need Clang C++ for the FleCSI Analyzer, even if they prefer Gnu C++ for their compilations. But this shouldn’t be a problem, as we anticipate that analysis and compilation will be (or can be) separate.

Beginnings

Nick did the initial work, which was clear and well-organized. It was, however, just a first cut, and wasn't entirely up-to-date with FleCSI's current code base.

My initial work has focused on three goals:

- **Clang**. Getting a feel for Clang and LLVM. While Clang's internals are allegedly far better than Gnu's, its learning curve is steep nonetheless.
- **FleCSI Macros**. Completing FleCSI Analyzer's ability to analyze calls to certain macros, particularly those in FleCSI's `data.h` and `execution.h`.
- **Organization**. I've completely refactored and reorganized the existing FleCSI Analyzer code, with future additions and improvements in mind.

Kudos to Nick for being extremely helpful whenever Clang presented me with a seemingly impenetrable brick wall!! Nick was awesome in the former way, Clang in the latter.

Usage (proposed; not yet implemented)

FleCSI Build:

Analysis:

Initialize “analysis info” file; currently using YAML

For each translation unit {

Perform each analysis stage {

- Preprocessor analysis:

For each element in the Abstract Syntax Tree (AST):

If the element was produced by a FleCSI macro {

Save relevant info to the “analysis info” file

}

- Other analysis stages, TBD

}

}

Compilation:

Possibly integrated with Analysis

Preprocessor Analysis

Perhaps a better term would be “macro analysis.”

This is the stage of analysis in which we’ll detect and analyze the use of certain macros in FleCSI codes.

Consider, for example, these FleCSI macros:

```
flecsi_register_task(task, nspace, processor, launch)

flecsi_execute_task(task, nspace, launch, ...)
```

The former “saves” the task by stringifying and hashing `task` and `nspace`. The latter retrieves the task in the same way.

Unfortunately, this means that a tpyo in `task` or `nspace`, in either macro call, will result in a run-time error instead of a compile-time error!

Preprocessor Analysis (continued)

Traditionally, in C and C++, preprocessing and compilation are seen as two distinct steps: the former a simple(-minded?), largely text-substitution step; the latter a real compilation.

Clang, it seems, actually **integrates the two steps**, or at least does so by default. The overall effect is still as defined in the Standard – as it must be.

The preprocessor/compiler integration allows us to make queries, at compile time, about a macro's definition, about where the macro is used, and about what code the macro produces.

Technically, we achieve the last bit indirectly. We recurse through the code's abstract syntax tree (AST), looking at each declaration, expression, etc. Each such construction possesses a "location" in the file. Cross-reference the location against data we've collected about the beginning and ending positions of each macro invocation, and we know whether or not to look more closely.

Step 1: Class Preprocessor

Consider our `Preprocessor` class. (Only part of its contents are shown.)

```
struct Preprocessor : public clang::PPCallbacks
{
   void MacroDefined(
      const clang::Token &,
      const clang::MacroDirective *const
   ) override;

   void MacroExpands(
      const clang::Token &,
      const clang::MacroDefinition &,
      const clang::SourceRange,
      const clang::MacroArgs *const
   ) override;
};
```

Plug this into Clang in a particular way. Then, `MacroDefined()` is called when a macro's *definition* is seen, while `MacroExpands()` is called wherever the macro is *invoked*. Here, we save textual information about each parameter to each macro of interest, and save locations (file positions) of each invocation.

Step 2: Class PreprocessorASTVisitor

Now consider our `PreprocessorASTVisitor` class. (Also shown only in part.)

```
class PreprocessorASTVisitor : public clang::RecursiveASTVisitor<PreprocessorASTVisitor>
{
   bool VisitVarDecl      (const clang::VarDecl       *const);
   bool VisitCallExpr     (const clang::CallExpr      *const);
   bool VisitTypeAliasDecl(const clang::TypeAliasDecl *const);
};
```

The base class endows `PreprocessorASTVisitor` with a traversal function with which we can visit the entire syntax tree of a file. We omit the details.

Each variable declaration triggers a `VisitVarDecl()` call. Each function-call expression triggers a `VisitCallExpr()` call. Each type-alias declaration (as in `using foo = bar`) triggers a `VisitTypeAliasDecl()` call.

Clang's `RecursiveASTVisitor<>` allows for a long list of `Visit*()` callbacks. We need to provide only what we care about. For our present macro-analysis purposes, the above functions suffice.

Preprocessor Analysis: Two Steps?

We just introduced two classes: Preprocessor, and PreprocessorASTVisitor. The former is used during preprocessing, to collect information about macro invocations. The latter is used for traversing the AST, during which we cross-reference with the information collected in the first step, and examine each AST element further if it's associated with a macro invocation.

Why isn't the second step just part of the first? (Or the first part of the second?)

Here's the basic idea. The first step provides only simple-minded information, stemming from the simple nature of preprocessing: text substitution.

The second step allows for proper C++ queries. Upon determining that a variable was produced by an invocation of `flecsi_register_data_client()`, for example, we can ask Clang if its type is derived from `data::data_client_t`.

Type information simply isn't available in the preprocessing stage.

Relevant macros, FYI

From **`data.hh`**:

```
flecsi_register_data_client     flecsi_get_handle           flecsi_get_mutator
flecsi_register_field           flecsi_get_client_handle    flecsi_get_global
flecsi_register_global          flecsi_register_color       flecsi_get_color
```

From **`execution.hh`**:

```
flecsi_register_task_simple         flecsi_execute_task_simple
flecsi_register_task                flecsi_execute_task
flecsi_register_mpi_task_simple     flecsi_execute_mpi_task_simple
flecsi_register_mpi_task            flecsi_execute_mpi_task
flecsi_register_global_object       flecsi_register_function
flecsi_set_global_object            flecsi_execute_function
flecsi_initialize_global_object     flecsi_function_handle
flecsi_get_global_object            flecsi_define_function_type
```

Not implemented: (some are helpers, some deprecated, some just not yet considered)

```
flecsi_register_program              __flecsi_internal_return_type
flecsi_register_top_level_driver     __flecsi_internal_arguments_type

flecsi_get_handles          flecsi_has_attribute        flecsi_put_all_handles
flecsi_get_handles_all      flecsi_is_at                flecsi_get_all_handles
                            flecsi_has_attribute_at
```

Relevant Files, FYI

Two existing Clang source files interact with the new FleCSI Analyzer code:

FrontendAction.cpp

ParseAST.cpp

New FleCSI Analyzer files (each with `.h` and `.cpp`):

FleCSIMisc (miscellaneous useful constructs)

FleCSIUtility (some simplified access to Clang capabilities)

FleCSIYAML (YAML structures for general use)

FleCSIPreprocessorYAML (YAML structures for macro analysis)

FleCSIPreprocessorASTVisitor (callbacks for macro analysis)

FleCSIPreprocessor (FleCSI Analyzer’s macro-analysis stage)

FleCSIAnalyzer (Overall FleCSI Analyzer)

Example: FleCSI Code

Consider this simple FleCSI example.

```
#include <execution.h>

namespace mynamespace {
   void foo() { }
   flecsi_register_task(foo, mynamespace, loc, single);
}

// We'll want to catch the fact that the following function
// is never registered, but we'll later try to execute it.
void bar(float) { }

namespace flecsi {
namespace execution {
   void driver(int argc, char **argv)
   {
      flecsi_execute_task(foo, mynamespace, single);
      flecsi_execute_task_simple(bar, single, float(1.234));
   }
}
}
```

Let's look at the YAML output that FleCSI Analyzer produces for this code.

Example: Relevant YAML Output

```
FleCSIRegisterTask:
  - macro:           flecsi_register_task
    file:            /home/staley/llvm/talk/x1.fcc
    line:            5
    task:            foo
    nspace:          mynamespace
    processor:       loc
    launch:          single
FleCSIExecuteTaskSimple:
  - macro:           flecsi_execute_task_simple
    file:            /home/staley/llvm/talk/x1.fcc
    line:            17
    task:            bar
    launch:          single
    varargs:
      - type:            float
        value:           'float(1.234)'
FleCSIExecuteTask:
  - macro:           flecsi_execute_task
    file:            /home/staley/llvm/talk/x1.fcc
    line:            16
    task:            foo
    nspace:          mynamespace
    launch:          single
    varargs:
```

Status, Issues, Etc.

YAML. We're just collecting information now, but not doing anything with it.

Brittleness. The macro processing is, in some sense, brittle. A slight change in a FleCSI macro might confuse the analyzer.

Build System. How should we integrate FleCSI Analyzer into the overall build system? We want the process to be simple for users. Remember, also, that the analyzer requires Clang, even if someone's compilation uses Gnu.

Prompt vs. Modular. Some errors (e.g. a `client_type` that isn't derived from `data::data_client_t`) are evident immediately; others (e.g. execution of an unregistered task) require examining all translation units first. I prefer to report errors as soon as they're known, but a case could perhaps be made to (1) write everything to YAML, then (2) report all errors, together, at a later time.

Features? Suggestions are welcome as to what FleCSI Analyzer should do! I know of just a few things, but the team collectively probably has more ideas.