Preprocessor-Aware Refactoring

Jeff Trull

12 May 2016

Outline

Refactoring is important
Most code is legacy code

- constantly changing requirements, tactics
- short-term focus restrains investment
- clean rewrite trades predictable cost for unknown, optimistically better, but mgmt hates risk
- all "human nature"
- this is our reality

Refactoring is important

Improving existing code is harder

- must learn to think like author(s) first
- often poor or no tests
- sometimes must refactor to make testable first
- Good news: doing this well may be more valuable (to employers, customers) than greenfield development

The Preprocessor gets in the way

- Macro substitution is a textual operation that can result in any program text whatsoever
- Conditional compilation hides parts of the code at compile time
- Generally what the compiler (and other tools) see and what the programmer has written are different.

The Preprocessor gets in the way
Macro Substitution

- Source of legacy idioms
  - global constants
    - what type is PI? MAXINT? NULL?
  - "helpful" global utilities e.g. min, max [1]
- Reproducible build issues - __DATE__, __TIME__, __TIMESTAMP__ [2]
- Barrier to refactoring: Scott Meyers blog [3]

```
#define ZERO 0
auto x = ZERO;
int *p = ZERO;
```

[1] "Using STL in Windows Program Can Cause Min/Max Conflicts" https://support.microsoft.com/en-us/kb/143208

[2] https://wiki.debian.org/ReproducibleBuilds/TimestampsFromCPPMacros

[3] "The Brick Wall of C++ Source Code Transformation" http://scottmeyers.blogspot.com/2015/11/the-brick-wall-of-c-source-code.html

The Preprocessor gets in the way
Conditional Compilation

- Static analysis (usually) only studies one configuration
  - `OPENSSL_NO_HEARTBEATS` [4]
- Accidental dead or unconditional code
  - `CONFIG_CPU_HOTPLUG` [5]
- Often there are better design idioms (e.g. template specialization for different cases)

[4] "Comments on a formal verification of PolarSSL" http://blog.regehr.org/archives/1261

[5] "How to avoid #ifdef bugs in the Linux kernel" https://www.linuxplumbersconf.org/2014/ocw/system/presentations/1863/original/rothberg.pdf

User tools

User tools
cpp2cxx

The fruit of a 2012 paper by Kumar, Sutton, and Stroustrup, "Rejuvenating C++ Programs through Demacrofication" [6]

- C++11/14 gives new options for macro replacement
  - Expression alias becomes `constexpr auto`; deduces type
  - Type alias becomes `using` statement
  - Parameterized type alias becomes `template<> using`

```
#define PTR_TYPE(T) T*
```

becomes

```
template <typename T>
using Ptr = T*;
```

[6] http://www.stroustrup.com/icsm-2012-demacro.pdf

User tools

cpp2cxx

- Parameterized expression becomes variadic function template
  - perfect forwarding permits argument type deduction

```
// macro
#define F(A1, ..., An) X
// C++11 declaration
template <typename T1, ..., typename Tn>
auto F(T1&& A1, ..., Tn&& An)
-> decltype(X)
{
    return X;
}
```

- Parameterized statement can become a similar function returning void
- resulting tool is cpp2cxx [7]
  - Actively maintained. Uses *both* Clang and Boost.Wave (?!)

[7] https://github.com/hiraditya/cpp2cxx

User tools
Clang tools

- clang-tidy "modernize nullptr" [8] handles the cases described by Scott Meyers
  - Replaces 0 and NULL assignment to pointers with `nullptr`
  - optionally handles user-selected macros as well
  - does not redefine the macro itself
- Clang Modularize [9]
  - Helps prepare for C++ "modules"
  - Looks for inconsistent macro definitions, among other things
  - Probably the most sophisticated PP/parser interaction tool I've seen

[8] `http://clang.llvm.org/extra/clang-tidy/checks/modernize-use-nullptr.html`
[9] `http://clang.llvm.org/extra/modularize.html`

User tools
Others

- cppcheck
  - hand-rolled parser etc.
  - does a surprisingly good job of handling configurations
- unifdef
  - Used to remove kernel-specific code from Linux code, and for understanding PP-heavy sources
  - `http://dotat.at/prog/unifdef/`

APIs

Boost.Wave

- Generic programming
- Lexer, preprocessor somewhat decoupled
- Preprocessor can do callbacks
- Spirit Classic
- Users
  - Imageworks (Sony Pictures) Open Shading Language [10]
  - ROSE (LLNL) Compiler Tools [11]

---

[10] `https://github.com/imageworks/OpenShadingLanguage`
[11] `http://rosecompiler.org/`

Boost.Wave
Lexer Usage


```
using namespace boost::wave;

using cpplexer_token_t = cpplexer::lex_token<>;
using cpplexer_iterator_t =
    cpplexer::lex_iterator<cpplexer_token_t>;

std::string cppstr{"struct Foo {};"};
auto cbeg = cppstr.begin();
cpplexer_iterator_t beg(cbeg, cppstr.end(),
                        cpplexer_token_t::position_type("fake.cpp"),
                        language_support(support_cpp|support_cpp0x));
cpplexer_iterator_t end;

for (auto tok = beg; tok != end; ++tok) {
    std::cout << tok.get_value();
}
```

Boost.Wave
Lexer Usage

```
using namespace boost::wave;

using cpplexer_token_t = cpplexer::lex_token<>;
using cpplexer_iterator_t =
    cpplexer::lex_iterator<cpplexer_token_t>;

std::string cppstr{"struct Foo {};"};
auto cbeg = cppstr.begin();
cpplexer_iterator_t beg(cbeg, cppstr.end(),
                        cpplexer_token_t::position_type("fake.cpp"),
                        language_support(support_cpp|support_cpp0x));
cpplexer_iterator_t end;

for (auto tok = beg; tok != end; ++tok) {
    std::cout << tok.get_value();
}
```

Boost.Wave
Lexer Usage


```
using namespace boost::wave;

using cpplexer_token_t = cpplexer::lex_token<>;
using cpplexer_iterator_t =
    cpplexer::lex_iterator<cpplexer_token_t>;

std::string cppstr{"struct Foo {};"};
auto cbeg = cppstr.begin();
cpplexer_iterator_t beg(cbeg, cppstr.end(),
                        cpplexer_token_t::position_type("fake.cpp"),
                        language_support(support_cpp|support_cpp0x));
cpplexer_iterator_t end;

for (auto tok = beg; tok != end; ++tok) {
    std::cout << tok.get_value();
}
```

Boost.Wave

Lexer Usage

```
using namespace boost::wave;

using cpplexer_token_t = cpplexer::lex_token<>;
using cpplexer_iterator_t =
    cpplexer::lex_iterator<cpplexer_token_t>;

std::string cppstr{"struct Foo {};"};
auto cbeg = cppstr.begin();
cpplexer_iterator_t beg(cbeg, cppstr.end(),
                        cpplexer_token_t::position_type("fake.cpp"),
                        language_support(support_cpp|support_cpp0x));
cpplexer_iterator_t end;

for (auto tok = beg; tok != end; ++tok) {
    std::cout << tok.get_value();
}
```

The Wave preprocessor wraps the lexer:

```
using policy_t =
    iteration_context_policies::load_file_to_string;
using context_t =
    context<std::string::const_iterator,
            cpplexer_iterator_t,
            policy_t,
            PPHooks>;

PPHooks   MyPPHooks;
context_t ctx(cppstr.begin(), cppstr.end(),
              "fake.cpp", MyPPHooks);
// many configuration methods on ctx here...
try {
    for (cpplexer_token const& tok : ctx) {
        std::cout << tok.get_value();
    }
} catch (preprocess_exception const& e) {
    std::cerr << "parse failed on line ";
    std::cerr << e.line_no() << "\n";
}
```

The Wave preprocessor wraps the lexer:

```
using policy_t =
    iteration_context_policies::load_file_to_string;
using context_t =
    context<std::string::const_iterator,
            cpplexer_iterator_t,
            policy_t,
            PPHooks>;

PPHooks   MyPPHooks;
context_t ctx(cppstr.begin(), cppstr.end(),
              "fake.cpp", MyPPHooks);
// many configuration methods on ctx here...
try {
    for (cpplexer_token const& tok : ctx) {
        std::cout << tok.get_value();
    }
} catch (preprocess_exception const& e) {
    std::cerr << "parse failed on line ";
    std::cerr << e.line_no() << "\n";
}
```

The Wave preprocessor wraps the lexer:

```
using policy_t =
    iteration_context_policies::load_file_to_string;
using context_t =
    context<std::string::const_iterator,
            cpplexer_iterator_t,
            policy_t,
            PPHooks>;

PPHooks   MyPPHooks;
context_t ctx(cppstr.begin(), cppstr.end(),
              "fake.cpp", MyPPHooks);
// many configuration methods on ctx here...
try {
    for (cpplexer_token const& tok : ctx) {
        std::cout << tok.get_value();
    }
} catch (preprocess_exception const& e) {
    std::cerr << "parse failed on line ";
    std::cerr << e.line_no() << "\n";
}
```

The Wave preprocessor wraps the lexer:

```
using policy_t =
    iteration_context_policies::load_file_to_string;
using context_t =
    context<std::string::const_iterator,
            cpplexer_iterator_t,
            policy_t,
            PPHooks>;

PPHooks   MyPPHooks;
context_t ctx(cppstr.begin(), cppstr.end(),
              "fake.cpp", MyPPHooks);
// many configuration methods on ctx here...
try {
    for (cpplexer_token const& tok : ctx) {
        std::cout << tok.get_value();
    }
} catch (preprocess_exception const& e) {
    std::cerr << "parse failed on line ";
    std::cerr << e.line_no() << "\n";
}
```

```
struct PPHooks : context_policies::default_preprocessing_hooks {

template <typename ContextT, typename TokenT>
bool found_directive(ContextT const &ctx,
                     TokenT const &directive);

template <typename ContextT, typename TokenT,
          typename ContainerT>
bool evaluated_conditional_expression(
    ContextT const &ctx,
    TokenT const& directive,
    ContainerT const& expression,
    bool expression_value);

};
```

```cpp
struct PPHooks : context_policies::default_preprocessing_hooks {

template <typename ContextT, typename TokenT>
bool found_directive(ContextT const &ctx,
                     TokenT const &directive);

template <typename ContextT, typename TokenT,
          typename ContainerT>
bool evaluated_conditional_expression(
    ContextT const &ctx,
    TokenT const& directive,
    ContainerT const& expression,
    bool expression_value);

};
```

Boost.Wave
Preprocessing Hooks


```
struct PPHooks : context_policies::default_preprocessing_hooks {

template <typename ContextT, typename TokenT>
bool found_directive(ContextT const &ctx,
                     TokenT const &directive);

template <typename ContextT, typename TokenT,
          typename ContainerT>
bool evaluated_conditional_expression(
    ContextT const &ctx,
    TokenT const& directive,
    ContainerT const& expression,
    bool expression_value);

};
```

```
struct PPHooks : context_policies::default_preprocessing_hooks {

template <typename ContextT, typename TokenT>
bool found_directive(ContextT const &ctx,
                     TokenT const &directive);

template <typename ContextT, typename TokenT,
          typename ContainerT>
bool evaluated_conditional_expression(
    ContextT const &ctx,
    TokenT const& directive,
    ContainerT const& expression,
    bool expression_value);

};
```

Clang libTooling
A Powerful Advantage

- easy access to Clang's Abstract Syntax Tree
- a nice API for performing code edits
- reformatting tools supplied
- used to write clang-tidy tools
- tightly coupled to other parts of Clang (e.g. source management)
- very OO

These tools are worth a presentation on their own. What follows is a quick summary; I found these talks particularly helpful:

- LLVM Developers Conference 2015, "An update on Clang-based C++ Tooling" [12]
- Richard Thomson C++Now 2014, "Create Your Own Refactoring Tool with Clang" [13]

[12] https://www.youtube.com/watch?v=1S2A0VWGOws
[13] https://www.youtube.com/watch?v=8PndHo7jjHk

Clang libTooling
PPCallbacks

- Similar to Boost.Wave Context Policy, but based on SourceLocation instead of tokens
- gives the full range of related text for directives, making it easy to identify related blocks
- tells you very little about skipped ranges - just their boundaries

Clang libTooling

PPCallbacks

```
struct MyPPHooks : clang::tooling::PPCallbacks
{

virtual void
If(SourceLocation      Loc,
   SourceRange         ConditionRange,
   ConditionValueKind ConditionValue
);

virtual void
Endif(SourceLocation  Loc,
      SourceLocation  IfLoc);

};
```

Clang libTooling
PPCallbacks

```
struct MyPPHooks : clang::tooling::PPCallbacks
{

virtual void
If(SourceLocation     Loc,
   SourceRange        ConditionRange,
   ConditionValueKind ConditionValue
);

virtual void
Endif(SourceLocation  Loc,
      SourceLocation  IfLoc);

};
```

Clang libTooling

PPCallbacks

```
struct MyPPHooks : clang::tooling::PPCallbacks
{

virtual void
If(SourceLocation      Loc,
   SourceRange         ConditionRange,
   ConditionValueKind  ConditionValue
);

virtual void
Endif(SourceLocation   Loc,
      SourceLocation   IfLoc);

};
```

A RefactoringTool instance is configured with *matchers* and their callbacks, and outputs *replacements*. It offers hooks to gain control at the start of parsing and perform actions, such as installing a `PPCallbacks` instance.

Clang libTooling
Matchers

- Help you find things in the AST
- Sort of a configurable visitor
- You can mark nodes of interest for processing by callbacks
- Three types:
    - Node
    - Narrowing
    - Traversal
- clang-query
    - sort of a CLI for matchers
- custom matchers

Clang libTooling

Matcher Example: Move Constructor

```
MoveCtorHandler  move_ctor_handler; // callback
using namespace clang::ast_matchers;
MatchFinder  finder;
finder.addMatcher(
   cxxConstructorDecl(                    // Node matcher
      isMoveConstructor()                 // Narrowing matcher
   ).bind("moveCtor"),                    // node of interest
   &move_ctor_handler);
```

Clang libTooling

Matcher Example: Move Constructor


```
MoveCtorHandler  move_ctor_handler; // callback
using namespace clang::ast_matchers;
MatchFinder  finder;
finder.addMatcher(
   cxxConstructorDecl(                  // Node matcher
      isMoveConstructor()               // Narrowing matcher
   ).bind("moveCtor"),                  // node of interest
   &move_ctor_handler);
```

```
MoveCtorHandler  move_ctor_handler; // callback
using namespace clang::ast_matchers;
MatchFinder  finder;
finder.addMatcher(
   cxxConstructorDecl(                // Node matcher
      isMoveConstructor()             // Narrowing matcher
   ).bind("moveCtor"),                // node of interest
   &move_ctor_handler);
```

Clang libTooling
Matcher Example: Move Constructor

```
MoveCtorHandler  move_ctor_handler; // callback
using namespace clang::ast_matchers;
MatchFinder  finder;
finder.addMatcher(
   cxxConstructorDecl(                // Node matcher
      isMoveConstructor()             // Narrowing matcher
   ).bind("moveCtor"),                // node of interest
   &move_ctor_handler);
```

```
MoveCtorHandler  move_ctor_handler; // callback
using namespace clang::ast_matchers;
MatchFinder  finder;
finder.addMatcher(
   cxxConstructorDecl(                  // Node matcher
      isMoveConstructor()               // Narrowing matcher
   ).bind("moveCtor"),                  // node of interest
   &move_ctor_handler);
```

```
MoveCtorHandler  move_ctor_handler; // callback
using namespace clang::ast_matchers;
MatchFinder  finder;
finder.addMatcher(
   cxxConstructorDecl(                   // Node matcher
      isMoveConstructor()                // Narrowing matcher
   ).bind("moveCtor"),                   // node of interest
   &move_ctor_handler);
```

```
const CXXConstructorDecl *decl =
     result.Nodes.getStmtAs<CXXConstructorDecl>("moveCtor");
auto loc = decl->getLocation();
if (ctx->getSourceManager().isInMainFile(loc)) {
    std::cout << "found a move constructor at "
              << loc.printToString(ctx->getSourceManager())
              << std::endl;
}
```

Clang libTooling

MatchCallback Example: inside MoveCtorHandler

```
const CXXConstructorDecl *decl =
     result.Nodes.getStmtAs<CXXConstructorDecl>("moveCtor");
auto loc = decl->getLocation();
if (ctx->getSourceManager().isInMainFile(loc)) {
    std::cout << "found a move constructor at "
              << loc.printToString(ctx->getSourceManager())
              << std::endl;
}
```

Clang libTooling

MatchCallback Example: inside MoveCtorHandler

```
const CXXConstructorDecl *decl =
    result.Nodes.getStmtAs<CXXConstructorDecl>("moveCtor");
auto loc = decl->getLocation();
if (ctx->getSourceManager().isInMainFile(loc)) {
   std::cout << "found a move constructor at "
             << loc.printToString(ctx->getSourceManager())
             << std::endl;
}
```

Clang libTooling

MatchCallback Example: inside MoveCtorHandler


```
const CXXConstructorDecl *decl =
     result.Nodes.getStmtAs<CXXConstructorDecl>("moveCtor");
auto loc = decl->getLocation();
if (ctx->getSourceManager().isInMainFile(loc)) {
    std::cout << "found a move constructor at "
              << loc.printToString(ctx->getSourceManager())
              << std::endl;
}
```

- A very basic edit: the replacement of some original text with new text
- If Replacements don't overlap, libTooling can intelligently combine them

```
Replacement  insert_at_start("foo.cpp", 0, 0,
                             "// New Header Comment");
Replacement  delete_something("foo.cpp", bad_code_start,
                              bad_code_length, "");
Replacement  replace_code(sourceManager, astNode,
                          "// new code");
```

Clang libTooling

Replacements

- A very basic edit: the replacement of some original text with new text
- If Replacements don't overlap, libTooling can intelligently combine them

```
Replacement  insert_at_start("foo.cpp", 0, 0,
                             "// New Header Comment");
Replacement  delete_something("foo.cpp", bad_code_start,
                              bad_code_length, "");
Replacement  replace_code(sourceManager, astNode,
                          "// new code");
```

Clang libTooling

Replacements

- A very basic edit: the replacement of some original text with new text
- If Replacements don't overlap, libTooling can intelligently combine them

```
Replacement  insert_at_start("foo.cpp", 0, 0,
                             "// New Header Comment");
Replacement  delete_something("foo.cpp", bad_code_start,
                              bad_code_length, "");
Replacement  replace_code(sourceManager, astNode,
                          "// new code");
```

Clang libTooling
Replacements

- A very basic edit: the replacement of some original text with new text
- If Replacements don't overlap, libTooling can intelligently combine them

```
Replacement  insert_at_start("foo.cpp", 0, 0,
                             "// New Header Comment");
Replacement  delete_something("foo.cpp", bad_code_start,
                              bad_code_length, "");
Replacement  replace_code(sourceManager, astNode,
                          "// new code");
```

```
using namespace clang::tooling;

CommonOptionsParser opt(argc, argv,
                        ToolingSampleCategory);
RefactoringTool     tool(opt.getCompilations(),
                         opt.getSourcePathList());
SourceFileCallbacks myCallbacks; // PPCallbacks

auto feFactory =
    newFrontendActionFactory(&finder, &myCallbacks).get();
if (int result = tool.run(feFactory)) {
    return result;
}
```

Clang libTooling
Putting it all together


```
using namespace clang::tooling;

CommonOptionsParser opt(argc, argv,
                        ToolingSampleCategory);
RefactoringTool     tool(opt.getCompilations(),
                         opt.getSourcePathList());
SourceFileCallbacks myCallbacks; // PPCallbacks

auto feFactory =
    newFrontendActionFactory(&finder, &myCallbacks).get();
if (int result = tool.run(feFactory)) {
    return result;
}
```

```
using namespace clang::tooling;

CommonOptionsParser opt(argc, argv,
                        ToolingSampleCategory);
RefactoringTool     tool(opt.getCompilations(),
                         opt.getSourcePathList());
SourceFileCallbacks myCallbacks; // PPCallbacks

auto feFactory =
    newFrontendActionFactory(&finder, &myCallbacks).get();
if (int result = tool.run(feFactory)) {
    return result;
}
```

Clang libTooling
Putting it all together

```
using namespace clang::tooling;

CommonOptionsParser opt(argc, argv,
                        ToolingSampleCategory);
RefactoringTool     tool(opt.getCompilations(),
                         opt.getSourcePathList());
SourceFileCallbacks myCallbacks; // PPCallbacks

auto feFactory =
    newFrontendActionFactory(&finder, &myCallbacks).get();
if (int result = tool.run(feFactory)) {
    return result;
}
```

Clang libTooling

Putting it all together

```
using namespace clang::tooling;

CommonOptionsParser opt(argc, argv,
                        ToolingSampleCategory);
RefactoringTool     tool(opt.getCompilations(),
                         opt.getSourcePathList());
SourceFileCallbacks myCallbacks; // PPCallbacks

auto feFactory =
    newFrontendActionFactory(&finder, &myCallbacks).get();
if (int result = tool.run(feFactory)) {
    return result;
}
```

Addressing Conditional Compilation

Conditional Compilation

Let's try applying our APIs to some interesting problems:

- Identifying the "presence condition" for each block of text
- Refactoring simple #ifdef/ifndef conditions into policy classes

Calculating Presence Conditions

- Split code into sections marked with the condition under which they are present
- Enables useful features:
  - Identify dead code
  - Identify code that appears conditional but is always present
  - Calculate source text under different assumptions
  - *Enumerate* all possible texts

Calculating Presence Conditions

The Problem to be Solved

```
#ifndef A
// section 1
#if (C > 10) && defined(B)
// section 2
#else
// section 3
#endif
#endif
```

Condition	Text
!defined(A)	// section 1
!defined(A) && (C>10) && defined(B)	// section 2
!defined(A) && ((C<=10) || !defined(B))	// section 3

Calculating Presence Conditions
Building Blocks Required

- A library that can represent conditional expressions, and combine and simplify them
- A lexical analyzer that handles C++ tokens
- A parser to recognize regular program text and preprocessor conditionals

What we need is an SMT solver.

SMT stands for *Satisfiability Modulo Theories*. Satisfiability, in turn, refers to finding assignments of values to variables such that an expression is true. For example, the expression

```
A && (X > 20) || !B && (Y <= 10)
```

is true (*satisfied*) for A true and X==21 - as well as many other values.

```
A && (X > 10) && (!A || (X == 9))
```

is not true for any choice of A and X - it is *unsatisfiable*.

SMT (in the form of its simpler cousin Boolean Satisfiability, or SAT) is *the* classic NP-complete problem. But solving it well regardless is enormously useful and so has received tons of research effort in the last decade. We will leverage that work.

Calculating Presence Conditions
Expressing Conditionals with CVC4

I picked CVC4 [14] based on a Google search but it's turned out well. Here's a quick tour:

```
ExprManager em;
SmtEngine smt(&em);
smt.setLogic("QF_LIA");                         // Linear Integer Arithmetic
Type boolean = em.booleanType();
Expr a = em.mkVar("A", boolean);                // bool defined(A)
Type integer = em.integerType();
Expr c = em.mkVar("C", integer);                // integer C
Expr expr =                                     // defined(A) && (C > 10)
    em.mkExpr(kind::AND, a,
              em.mkExpr(kind::GT, c,
                        em.mkConst(Rational(10))));
smt.assertFormula(                              // assume C == 20
    em.mkExpr(kind::EQUAL, c,
        em.mkConst(Rational(20))));
std::cout << "reduced expression is: ";
std::cout << smt.simplify(expr) << "\n";       // prints "A"
```

[14] http://cvc4.cs.nyu.edu/web/

Calculating Presence Conditions
Expressing Conditionals with CVC4

I picked CVC4 [14] based on a Google search but it's turned out well. Here's a quick tour:

```
ExprManager em;
SmtEngine smt(&em);
smt.setLogic("QF_LIA");                            // Linear Integer Arithmetic
Type boolean = em.booleanType();
Expr a = em.mkVar("A", boolean);                   // bool defined(A)
Type integer = em.integerType();
Expr c = em.mkVar("C", integer);                   // integer C
Expr expr =                                        // defined(A) && (C > 10)
    em.mkExpr(kind::AND, a,
              em.mkExpr(kind::GT, c,
                        em.mkConst(Rational(10))));
smt.assertFormula(                                 // assume C == 20
    em.mkExpr(kind::EQUAL, c,
        em.mkConst(Rational(20))));
std::cout << "reduced expression is: ";
std::cout << smt.simplify(expr) << "\n";           // prints "A"
```

[14] http://cvc4.cs.nyu.edu/web/

Calculating Presence Conditions
Expressing Conditionals with CVC4

I picked CVC4 [14] based on a Google search but it's turned out well. Here's a quick tour:

```
ExprManager em;
SmtEngine smt(&em);
smt.setLogic("QF_LIA");                          // Linear Integer Arithmetic
Type boolean = em.booleanType();
Expr a = em.mkVar("A", boolean);                 // bool defined(A)
Type integer = em.integerType();
Expr c = em.mkVar("C", integer);                 // integer C
Expr expr =                                      // defined(A) && (C > 10)
    em.mkExpr(kind::AND, a,
              em.mkExpr(kind::GT, c,
                        em.mkConst(Rational(10))));
smt.assertFormula(                               // assume C == 20
    em.mkExpr(kind::EQUAL, c,
        em.mkConst(Rational(20))));
std::cout << "reduced expression is: ";
std::cout << smt.simplify(expr) << "\n";        // prints "A"
```

[14] http://cvc4.cs.nyu.edu/web/

Calculating Presence Conditions

Expressing Conditionals with CVC4

I picked CVC4 [14] based on a Google search but it's turned out well. Here's a quick tour:

```
ExprManager em;
SmtEngine smt(&em);
smt.setLogic("QF_LIA");                        // Linear Integer Arithmetic
Type boolean = em.booleanType();
Expr a = em.mkVar("A", boolean);               // bool defined(A)
Type integer = em.integerType();
Expr c = em.mkVar("C", integer);               // integer C
Expr expr =                                    // defined(A) && (C > 10)
    em.mkExpr(kind::AND, a,
              em.mkExpr(kind::GT, c,
                        em.mkConst(Rational(10))));
smt.assertFormula(                             // assume C == 20
    em.mkExpr(kind::EQUAL, c,
        em.mkConst(Rational(20))));
std::cout << "reduced expression is: ";
std::cout << smt.simplify(expr) << "\n";       // prints "A"
```

[14] http://cvc4.cs.nyu.edu/web/

Calculating Presence Conditions

Expressing Conditionals with CVC4

I picked CVC4 [14] based on a Google search but it's turned out well. Here's a quick tour:

```
ExprManager em;
SmtEngine smt(&em);
smt.setLogic("QF_LIA");                          // Linear Integer Arithmetic
Type boolean = em.booleanType();
Expr a = em.mkVar("A", boolean);                 // bool defined(A)
Type integer = em.integerType();
Expr c = em.mkVar("C", integer);                 // integer C
Expr expr =                                      // defined(A) && (C > 10)
    em.mkExpr(kind::AND, a,
              em.mkExpr(kind::GT, c,
                        em.mkConst(Rational(10))));
smt.assertFormula(                               // assume C == 20
    em.mkExpr(kind::EQUAL, c,
        em.mkConst(Rational(20))));
std::cout << "reduced expression is: ";
std::cout << smt.simplify(expr) << "\n";        // prints "A"
```

[14] http://cvc4.cs.nyu.edu/web/

Calculating Presence Conditions

The Lexer

In order to use the Boost.Wave lexer with a Spirit V2 grammar we have to create wrappers for both the iterator and the token:

- both token and iterator need some special typedefs and methods
- also insert specializations into Spirit "customization points" to help us synthesize parsed results as strings

I will spare you the hacky details. . .

Calculating Presence Conditions
The Parser

As we collect text blocks, we must combine parsed conditions with their parent conditions:

- The condition for a text block is the logical AND of its own controlling condition and those of its parent
- #else or #elsif ANDs in negated conditions from "siblings"

Spirit rules are a nice fit for this task

Calculating Presence Conditions
Spirit Rule Anatomy

```
struct text_section {
    CVC4::Expr                  condition;
    std::vector<std::string> body;
};

using namespace boost::spirit;
qi::rule<Iterator,
         std::vector<text_section>(CVC4::Expr),
         skipper<Iterator>,                           // whitespace handling
         locals<CVC4::Expr>> cond_ifdef;
```

This rule type describes *inherited* attributes from nesting in the parent, the attribute *synthesized* by the rule, and a *local* attribute used to calculate the condition for the #else branch.

Calculating Presence Conditions

Spirit Rule Anatomy

```
struct text_section {
    CVC4::Expr               condition;
    std::vector<std::string> body;
};

using namespace boost::spirit;
qi::rule<Iterator,
         std::vector<text_section>(CVC4::Expr),
         skipper<Iterator>,                          // whitespace handling
         locals<CVC4::Expr>> cond_ifdef;
```

This rule type describes *inherited* attributes from nesting in the parent, the attribute *synthesized* by the rule, and a *local* attribute used to calculate the condition for the #else branch.

```
struct text_section {
    CVC4::Expr               condition;
    std::vector<std::string> body;
};

using namespace boost::spirit;
qi::rule<Iterator,
         std::vector<text_section>(CVC4::Expr),
         skipper<Iterator>,                          // whitespace handling
         locals<CVC4::Expr>> cond_ifdef;
```

This rule type describes *inherited* attributes from nesting in the parent, the attribute *synthesized* by the rule, and a *local* attribute used to calculate the condition for the #else branch.

Calculating Presence Conditions
Spirit Rule Anatomy

```
struct text_section {
    CVC4::Expr                 condition;
    std::vector<std::string> body;
};

using namespace boost::spirit;
qi::rule<Iterator,
         std::vector<text_section>(CVC4::Expr),
         skipper<Iterator>,                                 // whitespace handling
         locals<CVC4::Expr>> cond_ifdef;
```

This rule type describes *inherited* attributes from nesting in the parent, the attribute *synthesized* by the rule, and a *local* attribute used to calculate the condition for the #else branch.

```
struct text_section {
    CVC4::Expr               condition;
    std::vector<std::string> body;
};

using namespace boost::spirit;
qi::rule<Iterator,
         std::vector<text_section>(CVC4::Expr),
         skipper<Iterator>,                            // whitespace handling
         locals<CVC4::Expr>> cond_ifdef;
```

This rule type describes *inherited* attributes from nesting in the parent, the attribute *synthesized* by the rule, and a *local* attribute used to calculate the condition for the #else branch.

Calculating Presence Conditions
Spirit Rule Anatomy

```
struct text_section {
    CVC4::Expr                condition;
    std::vector<std::string> body;
};

using namespace boost::spirit;
qi::rule<Iterator,
         std::vector<text_section>(CVC4::Expr),
         skipper<Iterator>,                             // whitespace handling
         locals<CVC4::Expr>> cond_ifdef;
```

This rule type describes *inherited* attributes from nesting in the parent, the attribute *synthesized* by the rule, and a *local* attribute used to calculate the condition for the #else branch.

Calculating Presence Conditions

Putting It All Together


```
// create Spirit V2-compatible iterators from Wave lexer iterators
auto xbeg = make_tok_iterator(beg);
auto xend = make_tok_iterator(end);

vector<text_section> result;
bool pass = boost::spirit::qi::phrase_parse(xbeg, xend, fileparser,
                                             skipper<decltype(xbeg)>(),
                                                 result);
if (pass) {
    for (auto const& s : result) {
        if (smt.checkSat(s.condition) != CVC4::Result::SAT) {
            cout << "detected a dead code section with condition ";
            cout << smt.simplify(s.condition) << ":\n";
            copy(s.body.begin(), s.body.end(),
                 ostream_iterator<string>(cout, ""));
        }
    }
}
```

Calculating Presence Conditions

Putting It All Together


```cpp
// create Spirit V2-compatible iterators from Wave lexer iterators
auto xbeg = make_tok_iterator(beg);
auto xend = make_tok_iterator(end);

vector<text_section> result;
bool pass = boost::spirit::qi::phrase_parse(xbeg, xend, fileparser,
                                            skipper<decltype(xbeg)>(),
                                                result);

if (pass) {
    for (auto const& s : result) {
        if (smt.checkSat(s.condition) != CVC4::Result::SAT) {
            cout << "detected a dead code section with condition ";
            cout << smt.simplify(s.condition) << ":\n";
            copy(s.body.begin(), s.body.end(),
                 ostream_iterator<string>(cout, ""));
        }
    }
}
```

Calculating Presence Conditions
Putting It All Together


```
// create Spirit V2-compatible iterators from Wave lexer iterators
auto xbeg = make_tok_iterator(beg);
auto xend = make_tok_iterator(end);

vector<text_section> result;
bool pass = boost::spirit::qi::phrase_parse(xbeg, xend, fileparser,
                                            skipper<decltype(xbeg)>(),
                                               result);

if (pass) {
    for (auto const& s : result) {
        if (smt.checkSat(s.condition) != CVC4::Result::SAT) {
            cout << "detected a dead code section with condition ";
            cout << smt.simplify(s.condition) << ":\n";
            copy(s.body.begin(), s.body.end(),
                 ostream_iterator<string>(cout, ""));
        }
    }
}
```

Calculating Presence Conditions

Putting It All Together


```
// create Spirit V2-compatible iterators from Wave lexer iterators
auto xbeg = make_tok_iterator(beg);
auto xend = make_tok_iterator(end);

vector<text_section> result;
bool pass = boost::spirit::qi::phrase_parse(xbeg, xend, fileparser,
                                            skipper<decltype(xbeg)>(),
                                                result);

if (pass) {
    for (auto const& s : result) {
        if (smt.checkSat(s.condition) != CVC4::Result::SAT) {
            cout << "detected a dead code section with condition ";
            cout << smt.simplify(s.condition) << ":\n";
            copy(s.body.begin(), s.body.end(),
                 ostream_iterator<string>(cout, ""));
        }
    }
}
```

Calculating Presence Conditions
Putting It All Together


```
// create Spirit V2-compatible iterators from Wave lexer iterators
auto xbeg = make_tok_iterator(beg);
auto xend = make_tok_iterator(end);

vector<text_section> result;
bool pass = boost::spirit::qi::phrase_parse(xbeg, xend, fileparser,
                                            skipper<decltype(xbeg)>(),
                                                result);

if (pass) {
    for (auto const& s : result) {
        if (smt.checkSat(s.condition) != CVC4::Result::SAT) {
            cout << "detected a dead code section with condition ";
            cout << smt.simplify(s.condition) << ":\n";
            copy(s.body.begin(), s.body.end(),
                 ostream_iterator<string>(cout, ""));
        }
    }
}
```

Demo

Refactoring into Policies

Refactoring into Policies

The Problem to be Solved

Behavior and types for different configurations is sprinkled throughout the code.

```
char const* cstr = "foo";
#ifdef USE_QSTRING
    using string_t = QString;
    string_t s(cstr);
    s = s.toUpper();
#else
    using string_t = std::string;
    string_t s(cstr);
    std::transform(s.begin(), s.end(), s.begin(),
                   [](char c) { return std::toupper(c); });
#endif
```

Goal: isolate these variations in a *policy class* supplied as a template parameter.

- Access types with using statements
- Access code by calling static methods
- Conditional compilation only at point of instantiation, to choose policy

Desired classes:

```
template<bool UsingQString>
struct StringClass {
    // base template handles true case
    using string_t = QString;
    static void to_upper(string_t& s) {
        s = s.toUpper();
    }
};

template<>
struct StringClass<false> {
    using string_t = std::string;
    string_t s(cstr);
    static void to_upper(string_t& s) {
        std::transform(s.begin(), s.end(), s.begin(),
                       [](char c) { return std::toupper(c); });
    }
};
```

Usage:

```
// select policy class in a single place
#ifdef USE_QSTRING
using StringPolicy = StringClass<true>;
#else
using StringPolicy = StringClass<false>;
#endif

void my_fn() {
    using string_t = StringPolicy::string_t;
    string_t s("foo");           // chooses appropriate type
    StringPolicy::to_upper(s);   // calls appropriate code
    ...
}
```

Usage:

```
// select policy class in a single place
#ifdef USE_QSTRING
using StringPolicy = StringClass<true>;
#else
using StringPolicy = StringClass<false>;
#endif

void my_fn() {
    using string_t = StringPolicy::string_t;
    string_t s("foo");            // chooses appropriate type
    StringPolicy::to_upper(s);    // calls appropriate code
    ...
}
```

Refactoring into Policies

The Problem to be Solved

Usage:

```
// select policy class in a single place
#ifdef USE_QSTRING
using StringPolicy = StringClass<true>;
#else
using StringPolicy = StringClass<false>;
#endif

void my_fn() {
    using string_t = StringPolicy::string_t;
    string_t s("foo");           // chooses appropriate type
    StringPolicy::to_upper(s);   // calls appropriate code
    ...
}
```

Refactoring into Policies
Building Blocks Required

- A way to identify program text associated with a particular macro ifdef/ifndef
- A way to locate that text's AST subtree
    - Matchers can give us the typedefs and statements from there
- A way to determine the variables accessed and modified by that text
    - to determine the reference and const reference parameters of the static methods
- Code to integrate the above and produce edits

Refactoring into Policies

Identifying Conditional Text

```
template<bool Sense>    // defined or undefined?
struct PPActions : clang::PPCallbacks
{
    void Ifdef(clang::SourceLocation loc,
               clang::Token const& tok,
               clang::MacroDefinition const& md) override {
        // check for our target macro and sense
        if (tok.getIdentifierInfo()->getName().str() == mname_) {
            cond_starts_.emplace(loc, true);
            else_loc_ = std::experimental::nullopt;
        }
    }
    void Endif(clang::SourceLocation endifloc,
               clang::SourceLocation ifloc) override {
        // is this endif related to an ifdef/ifndef of interest?
        auto start_it = cond_starts_.find(ifloc);
        if (start_it != cond_starts_.end()) {
            // check sense, record range
...
    std::map<clang::SourceLocation, bool> cond_starts_;
```

Refactoring into Policies

Identifying Conditional Text

```
template<bool Sense>    // defined or undefined?
struct PPActions : clang::PPCallbacks
{
    void Ifdef(clang::SourceLocation loc,
               clang::Token const& tok,
               clang::MacroDefinition const& md) override {
        // check for our target macro and sense
        if (tok.getIdentifierInfo()->getName().str() == mname_) {
            cond_starts_.emplace(loc, true);
            else_loc_ = std::experimental::nullopt;
        }
    }
    void Endif(clang::SourceLocation endifloc,
               clang::SourceLocation ifloc) override {
        // is this endif related to an ifdef/ifndef of interest?
        auto start_it = cond_starts_.find(ifloc);
        if (start_it != cond_starts_.end()) {
            // check sense, record range
...
    std::map<clang::SourceLocation, bool> cond_starts_;
```

```
template<bool Sense>    // defined or undefined?
struct PPActions : clang::PPCallbacks
{
    void Ifdef(clang::SourceLocation loc,
               clang::Token const& tok,
               clang::MacroDefinition const& md) override {
        // check for our target macro and sense
        if (tok.getIdentifierInfo()->getName().str() == mname_) {
            cond_starts_.emplace(loc, true);
            else_loc_ = std::experimental::nullopt;
        }
    }
    void Endif(clang::SourceLocation endifloc,
               clang::SourceLocation ifloc) override {
        // is this endif related to an ifdef/ifndef of interest?
        auto start_it = cond_starts_.find(ifloc);
        if (start_it != cond_starts_.end()) {
            // check sense, record range
...
    std::map<clang::SourceLocation, bool> cond_starts_;
```

Clang's libASTMatchers doesn't provide a source range-based matcher, but we can make one:

```
AST_MATCHER_P(clang::Stmt,                statementInRange,
              clang::SourceRange,         range) {
    // is the statement node entirely within the supplied range?
    clang::SourceManager const& sm =
        Finder->getASTContext().getSourceManager();
    return !sm.isBeforeInTranslationUnit(Node.getLocStart(),
                                         range.getBegin()) &&
           !sm.isBeforeInTranslationUnit(range.getEnd(),
                                         Node.getLocEnd());
}
```

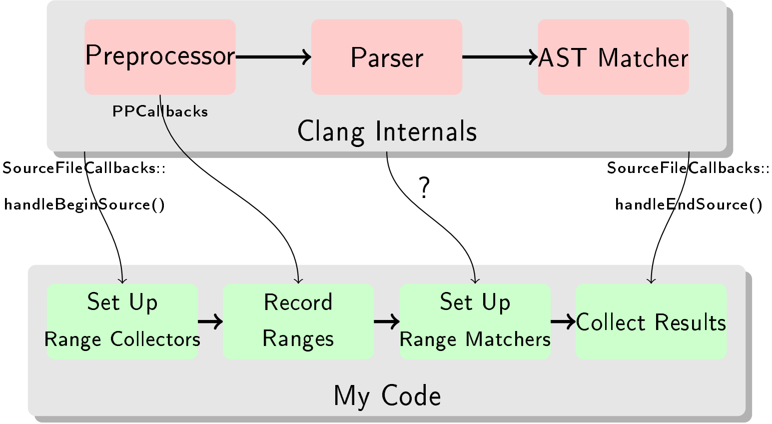
Refactoring into Policies
Locating an AST subtree from a SourceRange

Clang's libASTMatchers doesn't provide a source range-based matcher, but we can make one:

```
AST_MATCHER_P(clang::Stmt,                statementInRange,
              clang::SourceRange,         range) {
    // is the statement node entirely within the supplied range?
    clang::SourceManager const& sm =
        Finder->getASTContext().getSourceManager();
    return !sm.isBeforeInTranslationUnit(Node.getLocStart(),
                                         range.getBegin()) &&
           !sm.isBeforeInTranslationUnit(range.getEnd(),
                                         Node.getLocEnd());
}
```

Refactoring into Policies
Connecting ranges to matchers

This was a little tricky.



Refactoring into Policies

Connecting ranges to matchers

Hey, what's this?


clang 3.9.0svn

Main Page | Related Pages | Modules | Namespaces | Classes | Files

Class List | Class Index | Class Hierarchy | Class Members

clang 〉 ast_matchers 〉 MatchFinder 〉 ParsingDoneTestCallback 〉

clang::ast_matchers::MatchFinder::ParsingDoneTestCallback Class Reference abstract

Called when parsing is finished. Intended for testing only. More...

`#include <ASTMatchFinder.h>`

Public Member Functions

virtual	~ParsingDoneTestCallback ()
virtual void	run ()=0

Refactoring into Policies

Analyze Variables

Type definitions are fairly easy - we can have a matcher for those and move them to their specialization. Statements, which can reference or modify other variables, are more challenging. In this case we can apply a trick.

We can always:

- Create edits to the original source
- run those edits on an in-memory copy of the source
- run the compiler (and a tool) on that string with `runToolOnCode()`

How can we manipulate a source range to make it easier to identify variables used?

Annotate source with lambdas and analyze "captures"

```
auto expression_capture_0 = [&]() -> void {  // inserted
    s = s.toUpper();
}                                             // inserted
```

All variables referenced will be in the capture list in the AST. Must traverse lambda body to determine whether each is modified.

Refactoring into Policies

Analyze Variables: Process Flow

```
#ifdef FOO
    // true case
    v.push_back("bar");
#else
    // false case
    i = 42;
#endif
```

```
#ifdef FOO
auto _cond_statement_0 =
    [&]() {
    // true case
    v.push_back("bar");
};
_cond_statement_0();
#else
    // false case
    i = 42;
#endif
```

```
#ifdef FOO
    // true case
    v.push_back("bar");
#else
auto _cond_statement_0 =
    [&]() {
    // false case
    i = 42;
};
_cond_statement_0();
#endif
```

```
static void method_0(
   std::vector<std::string>& v,
   int& i
);
```

Refactoring into Policies
Sample Result


```
int main() {
#ifdef FOO
    typedef char i_t;
#else
    typedef int i_t;
    typedef char* string_t;
#endif

    i_t i;

#ifdef FOO
    i = '\0';
#else
    i = 1;
#endif
}
```

```
template<bool MacroDefined>
struct FOO_class {
    typedef char i_t;
    // static method TBD
};
template<>
struct FOO_class<false> {
    typedef int i_t;
    typedef char* string_t;
    // static method TBD
};
#ifdef FOO
    using FOO_t = FOO_class<true>;
#else
    using FOO_t = FOO_class<false>;
#endif
int main() {
    using i_t = FOO_t::i_t;
    i_t i;
    // statements TBD...
}
```

Refactoring into Policies

Watch Me Finish Up

Conclusion

- The preprocessor is a necessary evil
- It is also often misused or (especially with C++11/14) unnecessary
- We can write tools to remove some usage
- We can make tools more aware of it

Resources

- Garrido&Johnson "Analyzing Multiple Configurations of a C Program" (ICSM 2005)
  - Tool P-Cpp, implemented in CRefactory (Eclipse/Java)
- Sincero, "Efficient Extraction and Analysis of Preprocessor-Based Variability" (2010)
  - Found 4 dead code blocks in the Linux kernel
- Kästner "Partial Preprocessing C Code for Variability Analysis" (2011)
  - Rewrite all conditions in terms of user-controlled defines
  - don't handle integer expressions, just Boolean
  - Uses Java preprocessor `jcpp` and SAT solver `sat4j`
  - `https://github.com/joliebig/Morpheus`
- Gazillo and Grimm, "Parsing all of C by Taming the Preprocessor" (2012)
  - More Java :)
  - `http://cs.nyu.edu/xtc/`