Code transformation and analysis using Clang and LLVM

Static and Dynamic Analysis

Hal Finkel[1] and Gábor Horváth[2]

Computer Science Summer School 2017

[1] Argonne National Laboratory

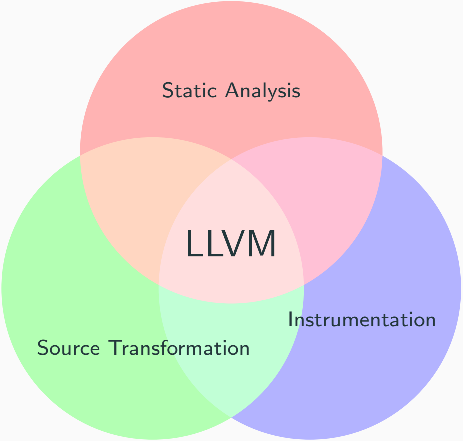
[2] Ericsson and Eötvös Loránd University

Table of contents

Introduction

Space of Techniques

During this set of lectures we'll cover a space of techniques for the analysis and transformation of code using LLVM. Each of these techniques have overlapping areas of applicability:



Space of Techniques

When to use source-to-source transformation:

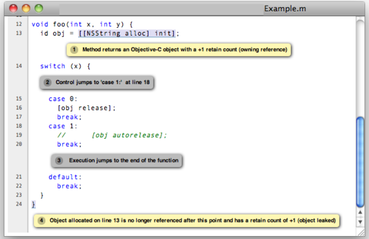
- When you need to use the instrumented code with multiple compilers.
- When you intend for the instrumentation to become a permanent part of the code base.

You'll end up being concerned with the textual formatting of the instrumentation if humans also need to maintain or enhance this same code.

Space of Techniques

When to use Clang's static analysis:

- When the analysis can be performed on an AST representation.
- When you'd like to maintain a strong connection to the original source code.
- When false negatives are acceptable (i.e. it is okay if you miss problems).



https://clang-analyzer.llvm.org/

Space of Techniques

When to use IR instrumentation:

- When the necessary conditions can be (or can only be) detected at runtime (often in conjunction with a specialized runtime library).
- When you require stronger coverage guarantees than static analysis.
- When you'd like to reduce the cost of the instrumentation by running optimizations before the instrumentation is inserted, after the instrumentation is inserted, or both.

You'll then need to decide whether to insert the instrumentation in Clang's CodeGen, later in LLVM's transformation pipeline, or both.

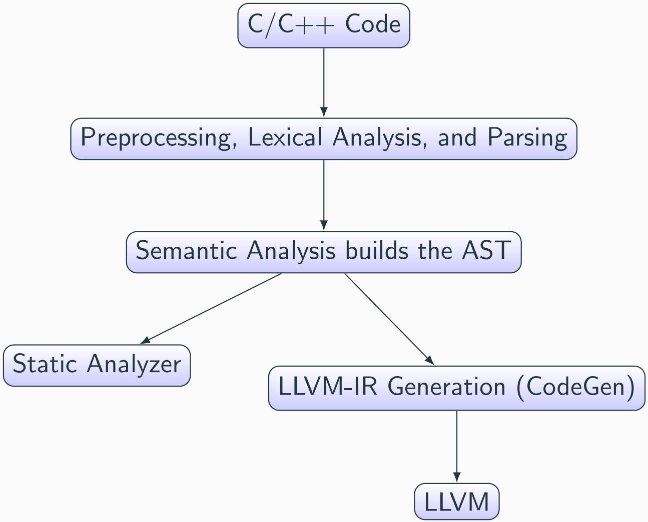
Space of Techniques

These techniques will be implemented in three different components of LLVM's infrastructure:

- **Clang**: Clang can perform static analysis, instrument the IR generated in Clang's CodeGen, and perform source-to-source transformations
- **LLVM**: IR-level transformations can add instrumentation before and/or after optimization
- **compiler-rt**: Instrumentation often requires corresponding components in the runtime library

Clang

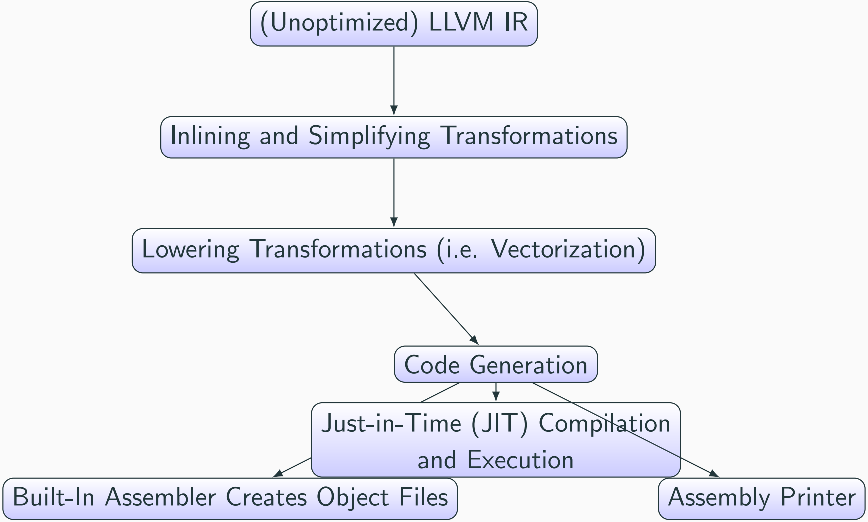
Clang:



C/C++ Code
Preprocessing, Lexical Analysis, and Parsing
Semantic Analysis builds the AST
Static Analyzer
LLVM-IR Generation (CodeGen)
LLVM

LLVM

LLVM:


(Unoptimized) LLVM IR
Inlining and Simplifying Transformations
Lowering Transformations (i.e. Vectorization)
Code Generation
Just-in-Time (JIT) Compilation and Execution
Built-In Assembler Creates Object Files
Assembly Printer

An Example

Let's following an example piece of code through the pipeline. If we have a file named test.cpp with:

```cpp
int main(int argc, char *argv[]) {
  return argc-1;
}
```

And we compile it with the command:

```
$ clang++ -v -O3 -o /tmp/test test.cpp
```

Note that we've used:

- Clang's driver's verbose mode
- Full optimizations

Optimization Levels

An aside on the meaning of LLVM's optimization levels...

- -O0: Only essential optimizations (e.g. inlining "always_inline" functions)
- -O1: Optimize quickly while losing minimal debugging information.
- -O2: Apply all optimization. Only apply transformations practically certain to produce better performance.
- -O3: Apply all optimizations including optimizations only **likely** to have a beneficial effect (e.g. vectorization with runtime legality checks).
- -Os: Same as -O2 but makes choices to minimize code size with minimal performance impact.
- -Oz: Same as -O2 but makes choices to minimize code size regardless of the performance impact.

An Example

```
$ clang++ -v -O3 -o /tmp/test test.cpp
clang version 4.0.1
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /path/to/llvm/4.0/bin
Found candidate GCC installation: /usr/lib/gcc/x86_64-redhat-linux/4.8.2
Found candidate GCC installation: /usr/lib/gcc/x86_64-redhat-linux/4.8.5
Selected GCC installation: /usr/lib/gcc/x86_64-redhat-linux/4.8.5
Candidate multilib: .;@m64
Candidate multilib: 32;@m32
Selected multilib: .;@m64
...
```

- By default, Clang selects the most-recent GCC in standard system paths (use `--gcc-toolchain=<path>` to pick a different toolchain explicitly).
- We're compiling 64-bit code. If you pass `-m32` it will select a 32-bit multilib configuration.

An Example

```
$ clang++ -v -O3 -o /tmp/test test.cpp
...
 "/path/to/llvm/4.0/bin/clang-4.0" -cc1 -triple x86_64-unknown-linux-gnu -
       emit-obj -disable-free -disable-llvm-verifier -discard-value-names -
       main-file-name test.cpp -mrelocation-model static -mthread-model
       posix -fmath-errno -masm-verbose -mconstructor-aliases -munwind-
       tables -fuse-init-array -target-cpu x86-64 -momit-leaf-frame-pointer
       -v -dwarf-column-info -debugger-tuning=gdb -resource-dir /path/to/
       llvm/4.0/bin/../lib/clang/4.0.1 -internal-isystem /usr/lib/gcc/x86_64
       -redhat-linux/4.8.5/../../../../include/c++/4.8.5 -internal-isystem /
       usr/lib/gcc/x86_64-redhat-linux/4.8.5/../../../../include/c++/4.8.5/
       x86_64-redhat-linux -internal-isystem /usr/lib/gcc/x86_64-redhat-
       linux/4.8.5/../../../../include/c++/4.8.5/backward -internal-isystem
       /usr/local/include -internal-isystem /path/to/llvm/4.0/bin/../lib/
       clang/4.0.1/include -internal-externc-isystem /include -internal-
       externc-isystem /usr/include -O3 -fdeprecated-macro -fdebug-
       compilation-dir /home/hfinkel -ferror-limit 19 -fmessage-length 150 -
       fobjc-runtime=gcc -fcxx-exceptions -fexceptions -fdiagnostics-show-
       option -fcolor-diagnostics -vectorize-loops -vectorize-slp -o
        /tmp/test-ca010f.o -x c++ test.cpp
...
```

- Clang's driver forks another Clang process to actually do the compilation itself. The first command-line argument is `-cc1`.
- These temporary files are removed after invocation. Use `-save-temps` for it to do otherwise.

An Example

```
$ clang++ -v -O3 -o /tmp/test test.cpp
...
ignoring nonexistent directory "/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-redhat-linux/4.8.5/../../../../include/c++/4.8.5
 /usr/lib/gcc/x86_64-redhat-linux/4.8.5/../../../../include/c++/4.8.5/x86_64-
       redhat-linux
 /usr/lib/gcc/x86_64-redhat-linux/4.8.5/../../../../include/c++/4.8.5/
       backward
 /usr/local/include
 /path/to/llvm/4.0/bin/../lib/clang/4.0.1/include
 /usr/include
End of search list.
...
```

The search path for include files is printed.

An Example

```
$ clang++ -v -O3 -o /tmp/test test.cpp
...
 "/usr/bin/ld" --hash-style=gnu --no-add-needed --eh-frame-hdr -m elf_x86_64
      -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o /tmp/test /usr/lib/gcc
      /x86_64-redhat-linux/4.8.5/../../../../lib64/crt1.o /usr/lib/gcc/
      x86_64-redhat-linux/4.8.5/../../../../lib64/crti.o /usr/lib/gcc/
      x86_64-redhat-linux/4.8.5/crtbegin.o -L/usr/lib/gcc/x86_64-redhat-
      linux/4.8.5 -L/usr/lib/gcc/x86_64-redhat-linux/4.8.5/../../../../
      lib64 -L/lib/../lib64 -L/usr/lib/../lib64 -L/usr/lib/gcc/x86_64-
      redhat-linux/4.8.5/../../.. -L/path/to/llvm/4.0/bin/../lib -L/lib -L/
      usr/lib /tmp/test-ca010f.o -lstdc++ -lm -lgcc_s -lgcc -lc -lgcc_s -
      lgcc /usr/lib/gcc/x86_64-redhat-linux/4.8.5/crtend.o /usr/lib/gcc/
      x86_64-redhat-linux/4.8.5/../../../../lib64/crtn.o
```

And, finally, the linker command is printed.

Clang

```
$ ls tools/clang/include/clang
Analysis         CodeGen    FrontendTool      Sema
ARCMigrate       Config     Index             Serialization
AST              Driver     Lex               StaticAnalyzer
ASTMatchers      Edit       module.modulemap  Tooling
Basic            Format     Parse
CMakeLists.txt   Frontend   Rewrite

$ ls tools/clang/lib
Analysis         CodeGen        Headers   Serialization
ARCMigrate       Driver         Index     StaticAnalyzer
AST              Edit           Lex       Tooling
ASTMatchers      Format         Parse
Basic            Frontend       Rewrite
CMakeLists.txt   FrontendTool   Sema
```

- We're going to start at the Driver...
- At end up at CodeGen.

Clang's main

A common question: Where is Clang's main function?

Clang's main function is in `driver.cpp` in:

```
$ ls tools/clang/tools/driver
cc1as_main.cpp  cc1_main.cpp  CMakeLists.txt  driver.cpp
```

```
...

int main(int argc_, const char **argv_) {
  llvm::sys::PrintStackTraceOnErrorSignal(argv_[0]);
  llvm::PrettyStackTraceProgram X(argc_, argv_);
  llvm::llvm_shutdown_obj Y; // Call llvm_shutdown() on exit.

  if (llvm::sys::Process::FixupStandardFileDescriptors())
    return 1;

  SmallVector<const char *, 256> argv;
  llvm::SpecificBumpPtrAllocator<char> ArgAllocator;
  std::error_code EC = llvm::sys::Process::GetArgumentVector(
      argv, llvm::makeArrayRef(argv_, argc_), ArgAllocator);
  if (EC) {
    llvm::errs() << "error: couldn't get arguments: " << EC.message() << '\n'
         ;
    return 1;
  }

  llvm::InitializeAllTargets();
  ...
```

Driver

- clang/lib
  - Driver
  - Frontend
  - Lex
  - Parse
  - Sema
  - AST
  - CodeGen

Driver

Command-line options for the driver are defined in `include/clang/Driver/Options.td`:

```
...
def fmath_errno : Flag<["-"], "fmath-errno">, Group<f_Group>, Flags<[
      CC1Option]>,
  HelpText<"Require␣math␣functions␣to␣indicate␣errors␣by␣setting␣errno">;
def fno_math_errno : Flag<["-"], "fno-math-errno">, Group<f_Group>;
def fbracket_depth_EQ : Joined<["-"], "fbracket-depth=">, Group<f_Group>;
def fsignaling_math : Flag<["-"], "fsignaling-math">, Group<f_Group>;
def fno_signaling_math : Flag<["-"], "fno-signaling-math">, Group<f_Group>;
def fjump_tables : Flag<["-"], "fjump-tables">, Group<f_Group>;
def fno_jump_tables : Flag<["-"], "fno-jump-tables">, Group<f_Group>, Flags<[
      CC1Option]>,
  HelpText<"Do␣not␣use␣jump␣tables␣for␣lowering␣switches">;
def fsanitize_EQ : CommaJoined<["-"], "fsanitize=">, Group<f_clang_Group>,
                   Flags<[CC1Option, CoreOption]>, MetaVarName<"<check>">,
                   HelpText<"Turn␣on␣runtime␣checks␣for␣various␣forms␣of␣
                         undefined␣"
                            "or␣suspicious␣behavior.␣See␣user␣manual␣for␣
                                  available␣checks">;
def fno_sanitize_EQ : CommaJoined<["-"], "fno-sanitize=">, Group<
      f_clang_Group>,
                      Flags<[CoreOption]>;
...
```

Driver

These flags are then accessed in various parts of the driver. I'll highlight two places in particular. One place is in Clang::ConstructJob in `lib/Driver/Tools.cpp`:

```
...
CmdArgs.push_back("-cc1");
...
CmdArgs.push_back("-mthread-model");
if (Arg *A = Args.getLastArg(options::OPT_mthread_model))
  CmdArgs.push_back(A->getValue());
else
  CmdArgs.push_back(Args.MakeArgString(getToolChain().getThreadModel()));

Args.AddLastArg(CmdArgs, options::OPT_fveclib);

if (!Args.hasFlag(options::OPT_fmerge_all_constants,
                  options::OPT_fno_merge_all_constants))
  CmdArgs.push_back("-fno-merge-all-constants");
...
```

This code composes the `-cc1` command line based on the driver's command-line arguments and other defaults (based on language, target, etc.).

Driver

A second place is in gnutools::Linker::ConstructJob in `lib/Driver/Tools.cpp`:

```
...
Args.AddAllArgs(CmdArgs, options::OPT_L);
Args.AddAllArgs(CmdArgs, options::OPT_u);

ToolChain.AddFilePathLibArgs(Args, CmdArgs);

if (D.isUsingLTO())
  AddGoldPlugin(ToolChain, Args, CmdArgs, D.getLTOMode() == LTOK_Thin, D);

if (Args.hasArg(options::OPT_Z_Xlinker__no_demangle))
  CmdArgs.push_back("--no-demangle");

bool NeedsSanitizerDeps = addSanitizerRuntimes(ToolChain, Args, CmdArgs);
...
```

This code composes the linker's command line. There's also a darwin::Linker::ConstructJob and so on for each supported toolchain.

Driver

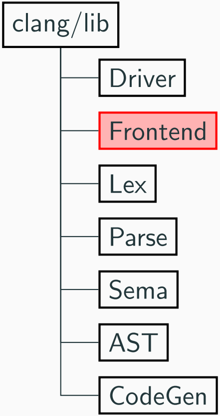
A third place is in `lib/Driver/SanitizerArgs.cpp`. This is relevant for adding new instrumentation and extending existing sanitizers.

```
  ...
  TsanAtomics = Args.hasFlag(options::OPT_fsanitize_thread_atomics,
                             options::OPT_fno_sanitize_thread_atomics,
                             TsanAtomics);
}

if (AllAddedKinds & CFI) {
  CfiCrossDso = Args.hasFlag(options::OPT_fsanitize_cfi_cross_dso,
                             options::OPT_fno_sanitize_cfi_cross_dso, false
                                    );
...
```

There's also generic code we'll look at later for detecting conflicts between requested sanitizers.

Frontend



Frontend

The flags composed by Clang::ConstructJob are consumed by ParseCodeGenArgs, etc. in `lib/Frontend/CompilerInvocation.cpp`:

```
static bool ParseCodeGenArgs(CodeGenOptions &Opts, ArgList &Args, InputKind
     IK,
                              DiagnosticsEngine &Diags,
                              const TargetOptions &TargetOpts) {
  ...
  Opts.DisableLLVMPasses = Args.hasArg(OPT_disable_llvm_passes);
  Opts.DisableLifetimeMarkers = Args.hasArg(OPT_disable_lifetimemarkers);
  Opts.DisableRedZone = Args.hasArg(OPT_disable_red_zone);
  Opts.ForbidGuardVariables = Args.hasArg(OPT_fforbid_guard_variables);
  Opts.UseRegisterSizedBitfieldAccess = Args.hasArg(
    OPT_fuse_register_sized_bitfield_access);
  Opts.RelaxedAliasing = Args.hasArg(OPT_relaxed_aliasing);
  Opts.StructPathTBAA = !Args.hasArg(OPT_no_struct_path_tbaa);
  Opts.DwarfDebugFlags = Args.getLastArgValue(OPT_dwarf_debug_flags);
  Opts.MergeAllConstants = !Args.hasArg(OPT_fno_merge_all_constants);
  ...
  Opts.CoverageMapping =
      Args.hasFlag(OPT_fcoverage_mapping, OPT_fno_coverage_mapping, false);
  Opts.DumpCoverageMapping = Args.hasArg(OPT_dump_coverage_mapping);
  Opts.AsmVerbose = Args.hasArg(OPT_masm_verbose);
  Opts.PreserveAsmComments = !Args.hasArg(OPT_fno_preserve_as_comments);
  ...
  Opts.DisableFPElim =
      (Args.hasArg(OPT_mdisable_fp_elim) || Args.hasArg(OPT_pg));
  ...
```

Frontend

Those code-generation options are defined in `include/clang/Frontend/CodeGenOptions.def`:

```
CODEGENOPT(DisableGCov          , 1, 0) ///< Don't run the GCov pass, for
      testing.
CODEGENOPT(DisableLLVMPasses    , 1, 0) ///< Don't run any LLVM IR passes to get
                                        ///< the pristine IR generated by the
                                        ///< frontend.
CODEGENOPT(DisableLifetimeMarkers, 1, 0) ///< Don't emit any lifetime markers
CODEGENOPT(ExperimentalNewPassManager, 1, 0) ///< Enables the new,
      experimental
                                             ///< pass manager.
CODEGENOPT(DisableRedZone       , 1, 0) ///< Set when -mno-red-zone is enabled.
CODEGENOPT(DisableTailCalls     , 1, 0) ///< Do not emit tail calls.
...
CODEGENOPT(NoImplicitFloat      , 1, 0) ///< Set when -mno-implicit-float is
      enabled.
CODEGENOPT(NoInfsFPMath         , 1, 0) ///< Assume FP arguments, results not +-
      Inf.
CODEGENOPT(NoSignedZeros        , 1, 0) ///< Allow ignoring the signedness of FP
       zero
CODEGENOPT(ReciprocalMath       , 1, 0) ///< Allow FP divisions to be
      reassociated.
CODEGENOPT(NoTrappingMath       , 1, 0) ///< Set when -fno-trapping-math is
      enabled.
CODEGENOPT(NoNaNsFPMath         , 1, 0) ///< Assume FP arguments, results not
      NaN.
...
```

Frontend

It is not uncommon to want to affect the predefined preprocessor macros. These are configured in `lib/Frontend/InitPreprocessor.cpp`:

```
static void InitializePredefinedMacros(const TargetInfo &TI,
                                       const LangOptions &LangOpts,
                                       const FrontendOptions &FEOpts,
                                       MacroBuilder &Builder) {
  // Compiler version introspection macros.
  Builder.defineMacro("__llvm__");  // LLVM Backend
  Builder.defineMacro("__clang__"); // Clang Frontend
  ...
  if (!LangOpts.MSVCCompat) {
    // Currently claim to be compatible with GCC 4.2.1-5621, but only if we'
        re
    // not compiling for MSVC compatibility
    Builder.defineMacro("__GNUC_MINOR__", "2");
    Builder.defineMacro("__GNUC_PATCHLEVEL__", "1");
    Builder.defineMacro("__GNUC__", "4");
    Builder.defineMacro("__GXX_ABI_VERSION", "1002");
  }
  ...
  // Standard conforming mode?
  if (!LangOpts.GNUMode && !LangOpts.MSVCCompat)
    Builder.defineMacro("__STRICT_ANSI__");
  ...
  if (TI.getPointerWidth(0) == 64 && TI.getLongWidth() == 64
     && TI.getIntWidth() == 32) {
    Builder.defineMacro("_LP64");
    Builder.defineMacro("__LP64__");
  }
  ...
```

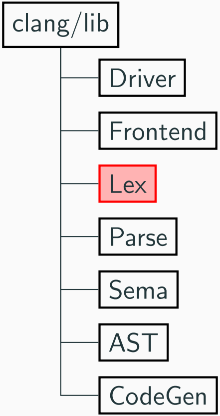
Frontend

Where do those language options come from? These are defined in `include/clang/Basic/LangOptions.def`:

```
...
LANGOPT(C99                    , 1, 0, "C99")
LANGOPT(C11                    , 1, 0, "C11")
...
LANGOPT(CPlusPlus              , 1, 0, "C++")
LANGOPT(CPlusPlus11            , 1, 0, "C++11")
...
LANGOPT(OpenMP                 , 32, 0, "OpenMP support and version of OpenMP (31,
       40 or 45)")
...
```

Note that the first number is the **number of bits** and the second is the default value!

Lex



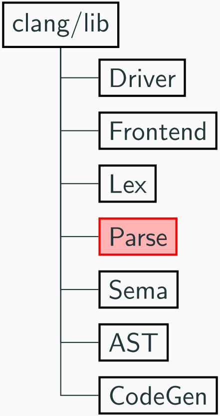
Lex

The main lexical analysis loop is in `lib/Lex/Lexer.cpp`. The preprocessor implementation is also part of the lexer. You'll probably not need to modify the lexer. You might come across a need to add a new keyword or some other kind of token. The tokens are defined in `include/clang/Basic/TokenKinds.def`:

```
...
KEYWORD(goto                        , KEYALL)
KEYWORD(if                          , KEYALL)
KEYWORD(inline                      , KEYC99|KEYCXX|KEYGNU)
KEYWORD(int                         , KEYALL)
KEYWORD(long                        , KEYALL)
KEYWORD(register                    , KEYALL)
...
ANNOTATION(pragma_openmp)
ANNOTATION(pragma_openmp_end)
ANNOTATION(pragma_loop_hint)
...
```

Note these "annotation" token definitions. These are used when transforming pragmas into token sequences that the parser can handle. If you'd like to add a new non-trivial pragma, then you'll probably need to add one of these tokens.

Parse



Parse

Clang uses a hand-written recursive-descent parser to parse C, C++, and several other dialects. Luckily, you probably won't need to modify the parser either. The most-likely parser modification you may need to make is adding a new pragma handler (or otherwise extending pragma parsing for loop hints, OpenMP, etc.). Pragmas are handled in `lib/Parse/ParsePragma.cpp`. For example:

```
...
struct PragmaLoopHintHandler : public PragmaHandler {
  PragmaLoopHintHandler() : PragmaHandler("loop") {}
  void HandlePragma(Preprocessor &PP, PragmaIntroducerKind Introducer,
                    Token &FirstToken) override;
};
...
void Parser::initializePragmaHandlers() {
  ...
  LoopHintHandler.reset(new PragmaLoopHintHandler());
  PP.AddPragmaHandler("clang", LoopHintHandler.get());
  ...
void Parser::resetPragmaHandlers() {
  ...
  PP.RemovePragmaHandler("clang", LoopHintHandler.get());
  LoopHintHandler.reset();
  ...
```

Parse

The pragma handler itself does some initial processing, inserts a special annotation token, and processes the token stream similar to how regular code is handled:

```
...
void PragmaLoopHintHandler::HandlePragma(Preprocessor &PP,
                                         PragmaIntroducerKind Introducer,
                                         Token &Tok) {
  // Incoming token is "loop" from "#pragma clang loop".
  Token PragmaName = Tok;
  SmallVector<Token, 1> TokenList;

  // Lex the optimization option and verify it is an identifier.
  PP.Lex(Tok);
  ...

  while (Tok.is(tok::identifier)) {
    ...
    // Generate the loop hint token.
    Token LoopHintTok;
    LoopHintTok.startToken();
    LoopHintTok.setKind(tok::annot_pragma_loop_hint);
    LoopHintTok.setLocation(PragmaName.getLocation());
    ...
  }
  ...
  PP.EnterTokenStream(std::move(TokenArray), TokenList.size(),
                      /*DisableMacroExpansion=*/false);
}
```

Parse

There's code in `lib/Parse/ParseStmt.cpp` that handles that annotation token and calls HandlePragmaLoopHint in `lib/Parse/ParsePragma.cpp`:

```
...
StmtResult
Parser::ParseStatementOrDeclarationAfterAttributes(StmtVector &Stmts,
         AllowedContsructsKind Allowed, SourceLocation *TrailingElseLoc,
         ParsedAttributesWithRange &Attrs) {
  ...
  case tok::annot_pragma_loop_hint:
    ProhibitAttributes(Attrs);
    return ParsePragmaLoopHint(Stmts, Allowed, TrailingElseLoc, Attrs);
...
StmtResult Parser::ParsePragmaLoopHint(StmtVector &Stmts,
  ...
  while (Tok.is(tok::annot_pragma_loop_hint)) {
    LoopHint Hint;
    if (!HandlePragmaLoopHint(Hint))
      continue;
...
bool Parser::HandlePragmaLoopHint(LoopHint &Hint) {
  assert(Tok.is(tok::annot_pragma_loop_hint));
  PragmaLoopHintInfo *Info =
      static_cast<PragmaLoopHintInfo *>(Tok.getAnnotationValue());
  ...
```

Parse

One final look at how the pragma information gets attached to the AST nodes as attribute information:

```
...
StmtResult Parser::ParsePragmaLoopHint(StmtVector &Stmts,
                                       AllowedContsructsKind Allowed,
                                       SourceLocation *TrailingElseLoc,
                                       ParsedAttributesWithRange &Attrs) {
  // Create temporary attribute list.
  ParsedAttributesWithRange TempAttrs(AttrFactory);

  // Get loop hints and consume annotated token.
  while (Tok.is(tok::annot_pragma_loop_hint)) {
    LoopHint Hint;
    if (!HandlePragmaLoopHint(Hint))
      continue;

    ArgsUnion ArgHints[] = {Hint.PragmaNameLoc, Hint.OptionLoc, Hint.StateLoc
        ,
                            ArgsUnion(Hint.ValueExpr)};
    TempAttrs.addNew(Hint.PragmaNameLoc->Ident, Hint.Range, nullptr,
                     Hint.PragmaNameLoc->Loc, ArgHints, 4,
                     AttributeList::AS_Pragma);
  }
  ...
  StmtResult S = ParseStatementOrDeclarationAfterAttributes(
      Stmts, Allowed, TrailingElseLoc, Attrs);

  Attrs.takeAllFrom(TempAttrs);
  return S;
}
...
```

AST

Let's go back to our example and look at some of Clang's AST. Given our test.cpp:

```cpp
int main(int argc, char *argv[]) {
  return argc-1;
}
```

We can look at a textual representation of as AST by running:

```
$ clang++ -fsyntax-only -Xclang -ast-dump test.cpp
```

```
TranslationUnitDecl 0x5b1a640 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x5b1abd0 <<invalid sloc>> <invalid sloc> implicit __int128_t '
      __int128'
| `-BuiltinType 0x5b1a8b0 '__int128'
...
`-FunctionDecl 0x5b4f5f8 <test.cpp:1:1, line:3:1> line:1:5 main 'int (int,
      char **)'
  |-ParmVarDecl 0x5b4f3c8 <col:10, col:14> col:14 used argc 'int'
  |-ParmVarDecl 0x5b4f4e0 <col:20, col:31> col:26 argv 'char **':'char **'
  `-CompoundStmt 0x5b4f780 <col:34, line:3:1>
    `-ReturnStmt 0x5b4f768 <line:2:3, col:15>
      `-BinaryOperator 0x5b4f740 <col:10, col:15> 'int' '-'
        |-ImplicitCastExpr 0x5b4f728 <col:10> 'int' <LValueToRValue>
        | `-DeclRefExpr 0x5b4f6e0 <col:10> 'int' lvalue ParmVar 0x5b4f3c8 '
              argc' 'int'
        `-IntegerLiteral 0x5b4f708 <col:15> 'int' 1
```

AST

Here's a loop:

```
int main(int argc, char *argv[]) {
  for (int i = 0; i < argc; ++i)
    return i;
}
```

```
...
`-FunctionDecl 0x65d9628 <test11.cpp:1:1, line:5:1> line:1:5 main 'int (int,
      char **)'
  |-ParmVarDecl 0x65d93f8 <col:10, col:14> col:14 used argc 'int'
  |-ParmVarDecl 0x65d9510 <col:20, col:31> col:26 argv 'char **':'char **'
  `-CompoundStmt 0x65d9960 <col:34, line:5:1>
    `-ForStmt 0x65d9928 <line:2:3, line:4:3>
      |-DeclStmt 0x65d97a8 <line:2:8, col:17>
      | `-VarDecl 0x65d9728 <col:8, col:16> col:12 used i 'int' cinit
      |   `-IntegerLiteral 0x65d9788 <col:16> 'int' 0
      |-<<<NULL>>>
      |-BinaryOperator 0x65d9840 <col:19, col:23> '_Bool' '<'
      | |-ImplicitCastExpr 0x65d9810 <col:19> 'int' <LValueToRValue>
      | | `-DeclRefExpr 0x65d97c0 <col:19> 'int' lvalue Var 0x65d9728 'i' '
            int'
      | `-ImplicitCastExpr 0x65d9828 <col:23> 'int' <LValueToRValue>
      |   `-DeclRefExpr 0x65d97e8 <col:23> 'int' lvalue ParmVar 0x65d93f8 '
            argc' 'int'
      |-UnaryOperator 0x65d9890 <col:29, col:31> 'int' lvalue prefix '++'
      | `-DeclRefExpr 0x65d9868 <col:31> 'int' lvalue Var 0x65d9728 'i' 'int'
      `-CompoundStmt 0x65d9908 <col:34, line:4:3>
        `-ReturnStmt 0x65d98f0 <line:3:5, col:12>
          `-ImplicitCastExpr 0x65d98d8 <col:12> 'int' <LValueToRValue>
            `-DeclRefExpr 0x65d98b0 <col:12> 'int' lvalue Var 0x65d9728 'i' '
                  int'
```

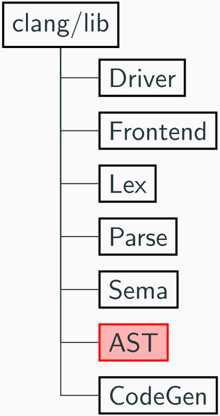
AST

And now with a pragma...

```
int main(int argc, char *argv[]) {
#pragma clang loop unroll(enable)
  for (int i = 0; i < argc; ++i)
    return i;
}
```

```
...
`-FunctionDecl 0x5346638 <test12.cpp:1:1, line:5:1> line:1:5 main 'int (int,
      char **)'
  |-ParmVarDecl 0x5346408 <col:10, col:14> col:14 used argc 'int'
  |-ParmVarDecl 0x5346520 <col:20, col:31> col:26 argv 'char **':'char **'
  `-CompoundStmt 0x53469c0 <col:34, line:5:1>
    `-AttributedStmt 0x53469a0 <<invalid sloc>, line:4:12>
      |-LoopHintAttr 0x5346980 <line:2:15, col:34> Implicit loop Unroll
            Enable
      | `-<<<NULL>>>
      `-ForStmt 0x5346948 <line:3:3, line:4:12>
        |-DeclStmt 0x53467e8 <line:3:8, col:17>
        | `-VarDecl 0x5346768 <col:8, col:16> col:12 used i 'int' cinit
        |   `-IntegerLiteral 0x53467c8 <col:16> 'int' 0
...
```

AST



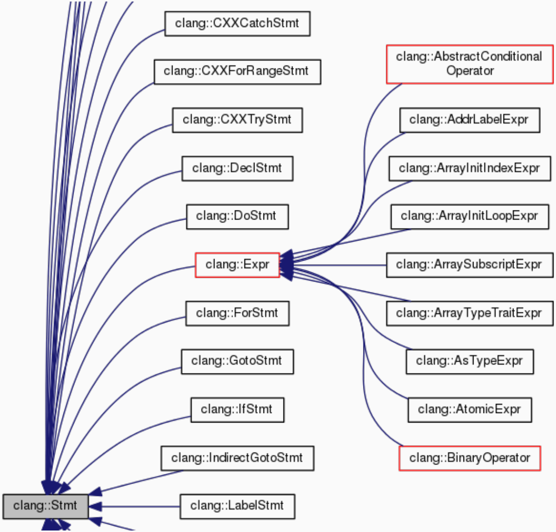
AST

Clang's Abstract Syntax Tree (AST) is a large class hierarchy representing essentially all supported source-level constructs. Importantly, Clang's AST is designed to be "faithful." This means that it tries to represent the input source constructs as closely as possible. Clang's AST is almost always treated as immutable (although some procedures, such as template instantiation, create new parts of the AST). CodeGen tends to walk the AST "as is" in order to generate LLVM IR.

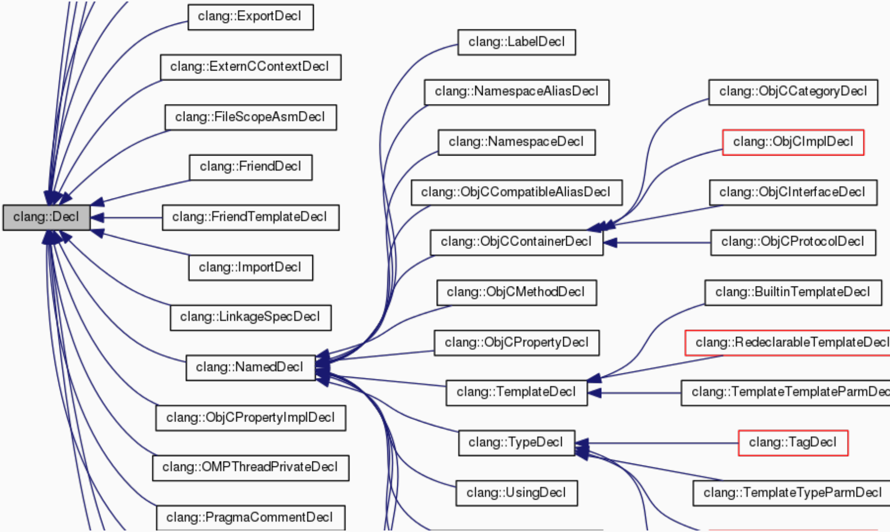
Clang's Doxygen-generated documentation is very useful for visualizing Clang's AST hierarchy:

`http://clang.llvm.org/doxygen` - `http://llvm.org/doxygen`

AST


clang::CXXCatchStmt
clang::CXXForRangeStmt
clang::CXXTryStmt
clang::DeclStmt
clang::DoStmt
clang::Expr
clang::ForStmt
clang::GotoStmt
clang::IfStmt
clang::IndirectGotoStmt
clang::LabelStmt
clang::Stmt
clang::AbstractConditional Operator
clang::AddrLabelExpr
clang::ArrayInitIndexExpr
clang::ArrayInitLoopExpr
clang::ArraySubscriptExpr
clang::ArrayTypeTraitExpr
clang::AsTypeExpr
clang::AtomicExpr
clang::BinaryOperator

AST



clang::ExportDecl
clang::ExternCContextDecl
clang::FileScopeAsmDecl
clang::FriendDecl
clang::Decl
clang::FriendTemplateDecl
clang::ImportDecl
clang::LinkageSpecDecl
clang::NamedDecl
clang::ObjCPropertyImplDecl
clang::OMPThreadPrivateDecl
clang::PragmaCommentDecl
clang::LabelDecl
clang::NamespaceAliasDecl
clang::NamespaceDecl
clang::ObjCCompatibleAliasDecl
clang::ObjCContainerDecl
clang::ObjCMethodDecl
clang::ObjCPropertyDecl
clang::TemplateDecl
clang::TypeDecl
clang::UsingDecl
clang::ObjCCategoryDecl
clang::ObjCImplDecl
clang::ObjCInterfaceDecl
clang::ObjCProtocolDecl
clang::BuiltinTemplateDecl
clang::RedeclarableTemplateDecl
clang::TemplateTemplateParmDec
clang::TagDecl
clang::TemplateTypeParmDecl

AST

The AST nodes are defined in header files in the `include/clang/AST`.

For example, from `include/clang/AST/Expr.h`:

```
class ImaginaryLiteral : public Expr {
  Stmt *Val;
public:
  ImaginaryLiteral(Expr *val, QualType Ty)
    : Expr(ImaginaryLiteralClass, Ty, VK_RValue, OK_Ordinary, false, false,
           false, false),
      Val(val) {}

  /// \brief Build an empty imaginary literal.
  explicit ImaginaryLiteral(EmptyShell Empty)
    : Expr(ImaginaryLiteralClass, Empty) { }

  const Expr *getSubExpr() const { return cast<Expr>(Val); }
  Expr *getSubExpr() { return cast<Expr>(Val); }
  void setSubExpr(Expr *E) { Val = E; }

  SourceLocation getLocStart() const LLVM_READONLY { return Val->getLocStart
      (); }
  SourceLocation getLocEnd() const LLVM_READONLY { return Val->getLocEnd(); }

  static bool classof(const Stmt *T) {
    return T->getStmtClass() == ImaginaryLiteralClass;
  }

  // Iterators
  child_range children() { return child_range(&Val, &Val+1); }
};
```

AST

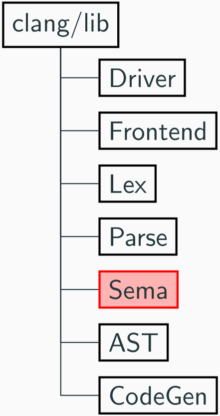
An AST visitor class is defined in `include/clang/AST/RecursiveASTVisitor.h`. There's also:

- include/clang/AST/CommentVisitor.h
- include/clang/AST/DeclVisitor.h
- include/clang/AST/EvaluatedExprVisitor.h
- include/clang/AST/StmtVisitor.h
- include/clang/AST/TypeLocVisitor.h
- include/clang/AST/TypeVisitor.h

A good example of how to use the visitor pattern to traverse the AST is in the AST pretty printers in `lib/AST/{Type,Stmt,Decl}Printer.cpp` files. For example:

```
...
void StmtPrinter::VisitWhileStmt(WhileStmt *Node) {
  Indent() << "while (";
  if (const DeclStmt *DS = Node->getConditionVariableDeclStmt())
    PrintRawDeclStmt(DS);
  else
    PrintExpr(Node->getCond());
  OS << ")\n";
  PrintStmt(Node->getBody());
}
...
```

Sema



Sema

After parsing some construct, the code in Parse calls an ActOn* method in Sema in order to build the part of the AST corresponding to that construct. The code in Sema performs semantics checks and, if the code is legal, constructs the corresponding AST node(s). For example:

```
StmtResult
Sema::ActOnDoStmt(SourceLocation DoLoc, Stmt *Body,
                  SourceLocation WhileLoc, SourceLocation CondLParen,
                  Expr *Cond, SourceLocation CondRParen) {
  assert(Cond && "ActOnDoStmt(): missing expression");

  CheckBreakContinueBinding(Cond);
  ExprResult CondResult = CheckBooleanCondition(DoLoc, Cond);
  if (CondResult.isInvalid())
    return StmtError();
  Cond = CondResult.get();

  CondResult = ActOnFinishFullExpr(Cond, DoLoc);
  if (CondResult.isInvalid())
    return StmtError();
  Cond = CondResult.get();

  DiagnoseUnusedExprResult(Body);

  return new (Context) DoStmt(Body, Cond, DoLoc, WhileLoc, CondRParen);
}
```

Sema

As an aside, let's look at some code that issues a diagnostic...

```
StmtResult Sema::ActOnForStmt(SourceLocation ForLoc, SourceLocation LParenLoc
    ,
                              Stmt *First, ConditionResult Second,
                              FullExprArg third, SourceLocation RParenLoc,
                              Stmt *Body) {
  if (Second.isInvalid())
    return StmtError();

  if (!getLangOpts().CPlusPlus) {
    if (DeclStmt *DS = dyn_cast_or_null<DeclStmt>(First)) {
      // C99 6.8.5p3: The declaration part of a 'for' statement shall only
      // declare identifiers for objects having storage class 'auto' or
      // 'register'.
      for (auto *DI : DS->decls()) {
        VarDecl *VD = dyn_cast<VarDecl>(DI);
        if (VD && VD->isLocalVarDecl() && !VD->hasLocalStorage())
          VD = nullptr;
        if (!VD) {
          Diag(DI->getLocation(), diag::err_non_local_variable_decl_in_for);
          DI->setInvalidDecl();
        }
      }
    }
  }
  ...
```

Sema

The diagnostic itself is defined in one of the `include/clang/Basic/Diagnostic*Kinds.td` files.

```
...
def err_non_local_variable_decl_in_for : Error<
  "declaration of non-local variable in 'for' loop">;
...
def warn_duplicate_attribute : Warning<
  "attribute %0 is already applied with different parameters">,
  InGroup<IgnoredAttributes>;
def warn_sync_fetch_and_nand_semantics_change : Warning<
  "the semantics of this intrinsic changed with GCC "
  "version 4.4 - the newer semantics are provided here">,
  InGroup<DiagGroup<"sync-fetch-and-nand-semantics-changed">>;
...
```

Where positional arguments are provided in iostreams style:

```
  ...
  S.Diag(Attr.getLoc(), diag::warn_duplicate_attribute) << Attr.getName();
  ...
```

CodeGen

- clang/lib
  - Driver
  - Frontend
  - Lex
  - Parse
  - Sema
  - AST
  - CodeGen

CodeGen

Clang's CodeGen generates LLVM IR walks the AST to generate (unoptimized) LLVM IR. It uses a visitor pattern. For example, in `lib/CodeGen/CGStmt.cpp`:

```
...
void CodeGenFunction::EmitStmt(const Stmt *S) {
  ...
  switch (S->getStmtClass()) {
  ...
  case Stmt::IfStmtClass:       EmitIfStmt(cast<IfStmt>(*S));
        break;
  case Stmt::WhileStmtClass:    EmitWhileStmt(cast<WhileStmt>(*S));
        break;
  case Stmt::DoStmtClass:       EmitDoStmt(cast<DoStmt>(*S));
        break;
  ...

void CodeGenFunction::EmitDoStmt(const DoStmt &S,
                                 ArrayRef<const Attr *> DoAttrs) {
  ...
  // Emit the body of the loop.
  llvm::BasicBlock *LoopBody = createBasicBlock("do.body");
  ....
  EmitBlockWithFallThrough(LoopBody, &S);
  {
    RunCleanupsScope BodyScope(*this);
    EmitStmt(S.getBody());
  }
  EmitBlock(LoopCond.getBlock());
...
```

CodeGen

Clang's CodeGen directory also contains the code that initializes LLVM's code-generator configuration, sets up and runs LLVM's pass manager, etc. To see how this works, look at the code in `lib/CodeGen/CodeGenAction.cpp`. This code calls EmitBackendOutput which is the entry point to the code in `lib/CodeGen/BackendUtil.cpp`.

Another important piece of code is in `lib/CodeGen/TargetInfo.cpp`. This file contains the Clang side of the (mostly undocumented) contract between Clang and LLVM regarding how LLVM constructs are lowered by LLVM's code generator in order to implement the calling conventions for each target.

Static Analysis with Clang

Static Analysis

Static Analysis is about analyzing your code without executing it. It can be used for a wide variety of purposes, but let's just concentrate on bug finding.

- The coverage of the analysis is not determined by the set of tests.
- The test cases might be in lack of some edge cases.
- No need for the runtime environment.
- Some properties can not be tested dynamically (e.g. coding conventions).
- There is no perfect static analysis.
    - Halting problem.
    - Rice's theorem.
    - Approximations.

Approximations

Can the value of `x` change?

```
int x = 4;
if (program P terminates on input I)
  ++x;
```

Undecidable!

We need to use approximations. The end result will not be perfect.

	Code is Incorrect	Code is Correct
Error Reported	True Positive	False Positive
No Error Reported	False Negative	True Negative

A common source of false positives: infeasible paths.

Consequences

Over-approximation

```
void f(int &x, bool change);

void g() {
  int y = 5;
  // Value of y is known.
  f(y, false);
  // Value of y is unknown.
}
```

Under-approximation

```
struct X {
  void m() const;
  ...
};
void f(X &x) {
  x.m(); // Assume: state is unchanged.
}
```

Both can introduce false positives or false negatives.

Static Analysis Tools in Clang

- Warnings
  - The first line of defense.
  - Support 'fixits'.
  - Fast and low false positive rate.
  - Applicable for most of the projects, no 3rd party API checks, coding convention related checks.
  - Based on AST, Preprocessor, Control Flow Graph.
- Clang-Tidy
  - Can be slower, can have higher false positive rate than a warning.
  - Checks for coding conventions, 3rd party APIs.
  - Support 'fixits'.
  - Matching the AST and inspecting the Preprocessor.
- Static Analyzer
  - Deep analysis: interprocedural, context sensitive, path sensitive.
  - Way slower than the compilation.
  - No 'fixits'.

Clang-Tidy

- In order to parse a code, you need the compilation commands.
- Like all Clang tools, Tidy supports the JSON compilation database.
  - CMake, Ninja can generate the database.
  - Scan-build-py can inspect the build and generate the database.
  - Using the `-MJ` flag of clang in your build system.
  - Specification: https://clang.llvm.org/docs/JSONCompilationDatabase.html

Invoking a clang tool:

```
./clang-tool filename -- compilation commands
# Or
./clang-tool -p /compilation/db/dir filename
```

AST Matchers

AST Matchers are embedded into C++ as a domain specific language to match subtrees of the AST. Better alternative to visitors.

Reference: http://clang.llvm.org/docs/LibASTMatchersReference.html

Basic categories of Matchers:

- Node matchers
- Narrowing matchers
- Traversal matchers

```
cxxOperatorCallExpr(
    anyOf(hasOverloadedOperatorName("="),
          hasOverloadedOperatorName("+=")),
    callee(cxxMethodDecl(ofClass(...))),
    hasArgument(1,
                expr(hasType(isInteger()),
                     unless(hasType(
                         isAnyCharacter())))
                     .bind("expr")),
    unless(isInTemplateInstantiation()))
```

Clang Query

- Invoked like any other clang tool.
- Interactive shell to experiment with Matchers.
- Completion on tab, history of commands, provided you have libEdit installed.

Clang Tidy Development Workflow

- Mock everything in your tests.
- Try to do most of your work with the Matchers.
- If something cannot be done by the Matcher, do it in the callback.
- Create 'fixits' if possible.
- Write documentation.
- Try it out on real world code.

Clang Tidy Checklist

What should you check before contributing back?

- Your check only triggers if the appropriate language standard is selected.
- Do not emit 'fixits' for code resulting from macro expansions.
- Do not emit problematic 'fixits' from implicit template instantiations.
- Default configuration option should give the least false positive answers.
- Cover all configuration options for your check in your tests.

Why are these problematic? You will see examples during the lab session.

Configuring Clang Tidy

Each check can have its own configuration options. See:
http://clang.llvm.org/extra/clang-tidy/#configuring-checks

- Configurations can be dumped.
- Projects can define options in a YAML file called `.clang-tidy`.
    - For each file, Tidy will look for the configuration file in the parent directories, thus subdirectories can have different settings.
- Additional check specific configurations can be passed from the command line.
- For Clang Tidy specific options see the `-help` menu.
- Clang Tidy can display compiler warning messages and Static Analyzer messages but the outputs are limited to plain text.

Source-to-Source Transformation

Clang Tidy can: replacement, remove, insert text. See:
https://clang.llvm.org/doxygen/classclang_1_1FixItHint.html

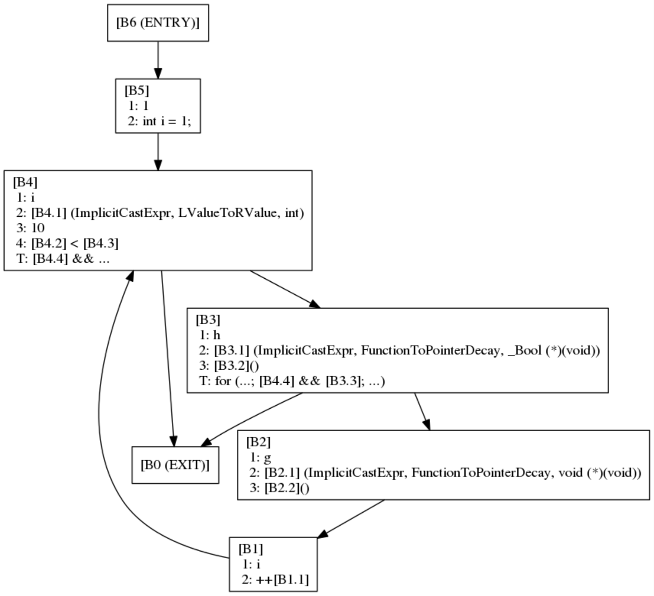
- The AST is not transformed and/or pretty printed.
    - The AST has many invariants, hard to transform.
    - Not designed to be pretty printed, preserving comments/white spaces/macros are suboptimal.
- Conflicting edits will cause problems, nonconflicting duplicates are removed.
- We do not want to break the compilation.
- We definitely do not want to break the semantics of the code.
- Headers can cause problems, the solution:
    - Accumulate all 'fixits' for all translation units, deduplicate, apply all.
- Some 'fixits' are dependent on each other, the solution:
    - Atomic 'fixit' sets; if any of them fails, none of them will be applied.
- Some cleanups are done by Tidy (e.g.: removing colon after removing the last element from the init list), and formatting optionally.

The control flow graph

```
bool h();
void g();

void f() {
  for(int i = 1; i < 10 && h(); ++i)
    g();
}
```

The control flow graph



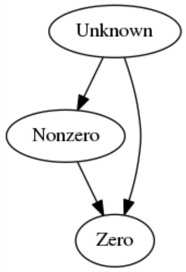
Flow-Sensitive Analysis

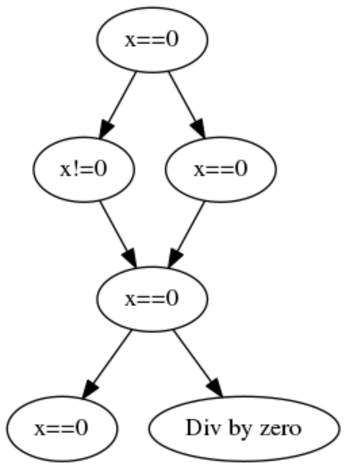
```c
int x = 0;
if (z)
    x = 5;
if (z)
    y = 10 / x;
```

Find division by zero by walking the cfg! What to do on merge points?

Flow-Sensitive Analysis

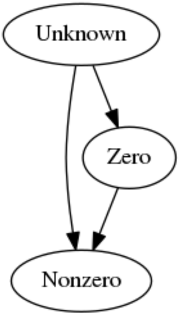
```
int x = 0;
if (z)
    x = 5;
if (z)
    y = 10 / x;
```

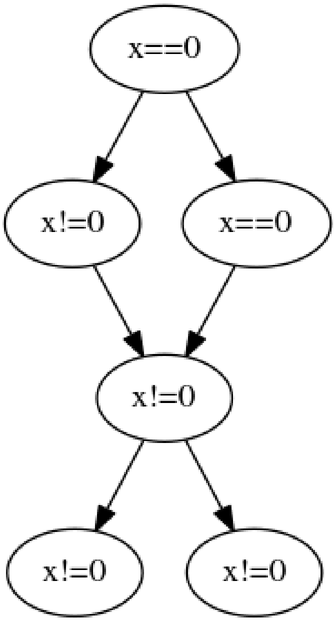




Flow-Sensitive Analysis

```
int x = 0;
if (z)
    x = 5;
if (!z)
    y = 10 / x;
```





Flow-Sensitive vs Path-Sensitive Analysis

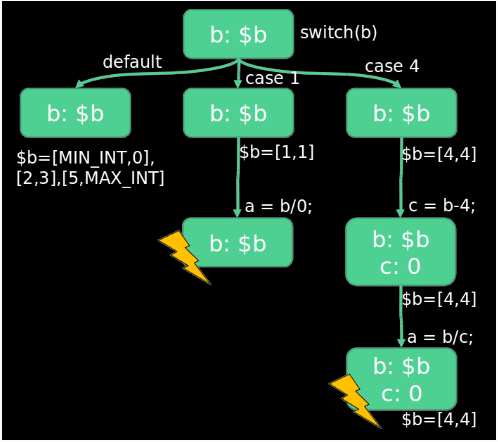
- The flow-sensitive analysis is not precise enough! We either have false positives or false negatives.
- Flow-sensitive compiler warnings favor false negatives.
- Flow-sensitive algorithms can be fast.
- What if we do not merge the states? We call such algorithms path-sensitive. The resulting model is exponential in the branching factor, but more precise.

Symbolic execution

- Symbolic execution is a path-sensitive algorithm.
- Interprets the source code with abstract semantics instead of the actual.
- When it encounters an unknown value (e.g. a value read from the user), represents it as a symbol.
- Records the constraints on the symbols based on the conditions that were visited.
- The Clang Static Analyzer uses symbolic execution.
- Let's analyze some code using symbolic execution!

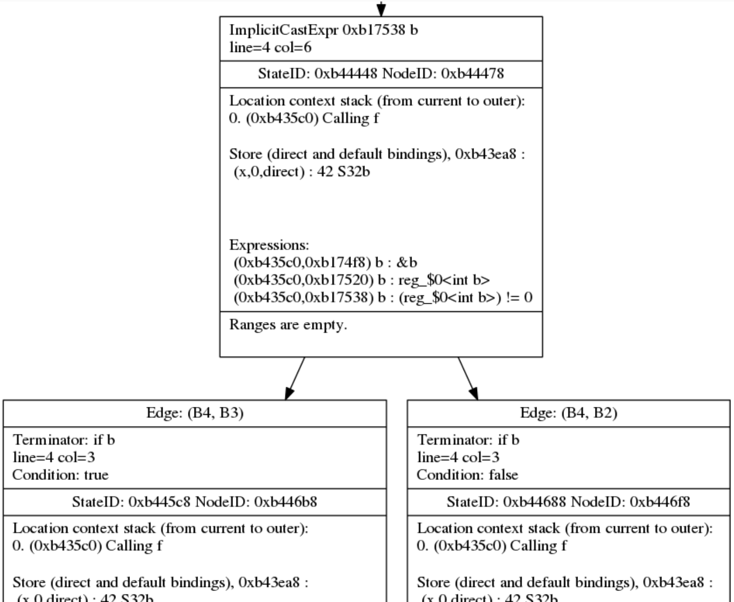
Symbolic execution

```
void f(int b) {
  int a,c;
  switch (b){
    case 1:
      a = b / 0;
      break;
    case 4:
      c = b - 4;
      a = b / c;
      break;
  }
}
```



Exploded Graph

Exploded Graph is a data structure that supports symbolic execution.

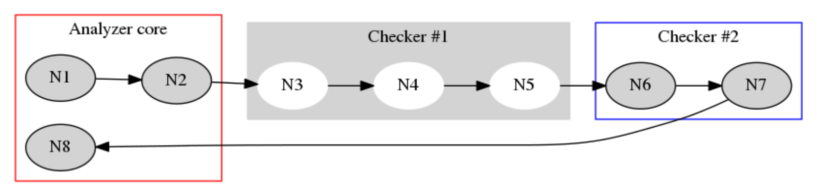


Exploded Graph

- Each node is a **Program Point** and **Symbolic State** pair.
- A Program Point is a location in the source code with a stack frame.
- A Symbolic State is a set of Program States.
- Each edge is a change in the Symbolic State or the Program Point.
- If we see an already visited Exploded Node we do not continue the analysis.
- The size of this graph can be huge.

Checker Collaboration

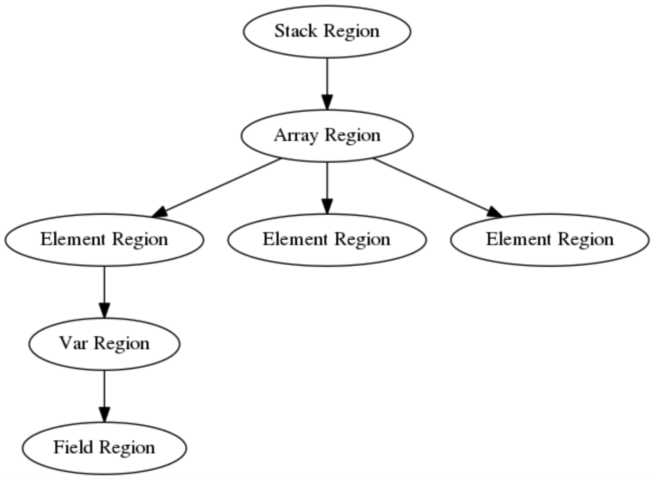
The checkers and the Static Analyzer core participate in building the exploded graph.



Memory regions

Memory is represented using hierarchical regions.

```
A a[10];
a[5].f
```



Program State

The program state consists of three parts:

- Store
    - Mapping between memory regions and symbolic values
    - A symbolic value can be a symbolic memory region
- Environment
    - Mapping between source code expressions and symbolic values
    - Only the currently evaluated expressions are part of this mapping
- Generic Data Map
    - The store of checker specific state
    - Can be a value, set, or a map from symbols or memory regions to other data.

Infeasible Path

To have low false positive rate, it is crucial to not analyze infeasible path. Every report on an infeasible path is a false positive.

```
int x = 0;
if (z)
    x = 5;
if (!z)
    y = 10 / x;
```

If the first branch is taken, the constraint that z is true is recorded and the second branch will not be taken.

For complicated expressions, the constraint manager might not be able to eliminate the infeasible path resulting in false positives.

Z3 can be used as a more precise, but slower, constraint solver in the static analyzer.

Constraint Manager

There is no way to query/enumerate the constraints on a symbol.

If you want to answer questions about a symbolic value, you need to build a new symbolic expression and evaluate it with the constraint manager.

You can also add new constraints.

```
SValBuilder &svalBuilder =
  C.getSValBuilder();
DefinedOrUnknownSVal zero =
  svalBuilder.makeZeroVal(Ty);

ProgramStateRef stateNull, stateNonNull;
std::tie(stateNull, stateNonNull) =
  state->assume(svalBuilder.evalEQ(state,
                                   val,
                                   zero));
```

Interprocedural Analysis

Errors might span across function calls. It is important to have interprocedural analysis.

```c
int foo() {
  return 0;
}
void bar() {
  int x = 5 / foo();
}
```

Context Sensitivity

Sometimes we need the calling context to find the error. If we can use the information from the call site, the analysis is context sensitive.

```
int foo(int x) {
  return 42 - x;
}
void bar() {
  int x = 5 / foo(42);
}
```

The Clang Static Analyzer conceptually inlines the calls to get context sensitivity.

Inlining

What to do with recursion or long call chains? After a call stack depth limit the analyzer gives up with inlining functions.

How to start the analysis? The function where the analysis starts is called the top level function. It has no calling context.

The analyzer creates a topological order of the functions. If a function was already inlined it might not be analyzed as the top level.

Invalidation

```
int foo(int *x);
int bax(int *x);
void bar(int y) {
  if (y == 42) {
    // Value of y?
    foo(&y);
    // Value of y?
  }
  int *p = new int;
  baz(p);
  // Is there a leak?
}
```

Sometimes we need to invalidate the information we collected so far.

Dead Symbols

When can we be sure that there is a leak?

```
void bar(int y) {
  int *p = new int;
  // ...
  p = 0;
  // ...
}
```

When we know that the memory is not freed and p is no longer used.

The symbol representing the value of p is dead.

Callbacks

The checkers can subscribe to events during symbolic execution. They are visiting (while building) the nodes of the exploded graph.

```
void checkPreCall(const CallEvent &Call,
                  CheckerContext &C);
void checkDeadSymbols(SymbolReaper &SR,
                      CheckerContext &C);
void checkBind(SVal Loc, SVal Val,
               const Stmt *S,
               CheckerContext &);
void checkEndAnalysis(ExplodedGraph &G,
                      BugReporter &BR,
                      ExprEngine &Eng);
ProgramStateRef checkPointerEscape(
    ProgramStateRef State,
    const InvalidatedSymbols &Escaped,
    const CallEvent *Call,
    PointerEscapeKind Kind);
// ...
```

Checker Specific Program Sates

Checkers might extend the program state with checker specific data. See: https://clang-analyzer.llvm.org/checker_dev_manual.html#extendingstates

```
REGISTER_MAP_WITH_PROGRAMSTATE(ExampleDataType,
                               SymbolRef, int)

ProgramStateRef state;
SymbolRef Sym;
// ...
int curVal = state->get<ExampleDataType>(Sym);
// ...
ProgramStateRef newState =
  state->set<ExampleDataType>(Sym, newValue);

context.addTransition(newState /*, Pred*/);
```

Sinks

What to do when the analyzer finds an issue?

- Is this issue critical like division by zero? Terminate the analysis on that path. (Generate a Sink node.)
- The issue is not critical? Report an error and continue the analysis.
- When a precondition of a function is violated, do not report errors for violating the post-conditions of the function
- Add the appropriate bug path visitors

```
int *_Nonnull f(int *_Nonnull);
```

Bug Path Visitors

After the analyzer found the bug, it should make a report easy to understand.

With Bug Path Visitors, you can add additional notes to the bug path.

- Compare the program states pairwise along the path.
- Detect relevant changes in the state and emit a note.
    - Division by zero? What is the source of the value?
    - Use of a moved from object? Where did the move happen?

Debugging a Checker

The Static Analyzer is complex, this makes debugging a challenge. It is important to have minimal test cases. There are also some helpful tools to inspect the behavior of the analyzer. You can also use common methods, like attaching a debugger.

```
# Analyzer related options
clang -cc1 -help | grep "analyzer"
# Checker list
clang -cc1 -analyzer-checker-help
# Analyzing a file with a checker
clang -cc1 -analyze -analyzer-checker=core.
    DivideZero \
  a.c
clang -cc1 -analyze -analyzer-checker=core a.c \
  -analyzer-display-progress -analyze-function=
      foo
clang -cc1 -analyze \
  -analyzer-checker=debug.ViewCFG a.c
clang -cc1 -analyze \
  -analyzer-checker=debug.ViewExplodedGraph a.c
```

Annotate Custom Assertions

The analyzer can understand standard assertions.

It is important to analyze debug builds, it helps the precision.

Custom assertions can be a source of a problem unless they are annotated.

```
void customAssert();
int foo(int *b) {
  if (!b)

        1  Assuming 'b' is null

     2  Taking true branch

    customAssert();
  return *b;

           3  Dereference of null pointer (loaded from variable 'b')

}
```

Inline Defensive Checks

An important heuristic to suppress false positives.

```c
void f(int *p) {
  if (!p)
    return;
  // ...
}

void g(int *p) {
  f(p);
  *p = 3;
}
```

We should not report null dereference warning in this case!

Size of the Exploded Graph

It is not feasible to build and traverse the whole exploded graph.

There are some heuristics to keep the size reasonable:

- Limit on the number of visited nodes
- Limit on the number of visits to the same basic block in the CFG
- Limit on the max size of the stack of the inlined functions
- Limit on the inlining of large functions
- ...

Cross Translation Unit Analysis

The Clang Static Analyzer does not support CTU currently.

```cpp
int bar(int &i);

void f(int j) {
  // ...
  if (j == 42)
    x = 5 / bar(j);
  int k = j + 1;
  // ...
}
```

```cpp
int bar(int &i) {
  return i - 42;
}
```

We loose information and true positive results.

We may also get false positives.

Partial CTU Analysis

There are some supported ways to reason about properties of the program across the translation unit boundaries.

- Type system
- Annotations like nullability
- Body farm (hand written AST pieces)
- Checkers modeling library functions

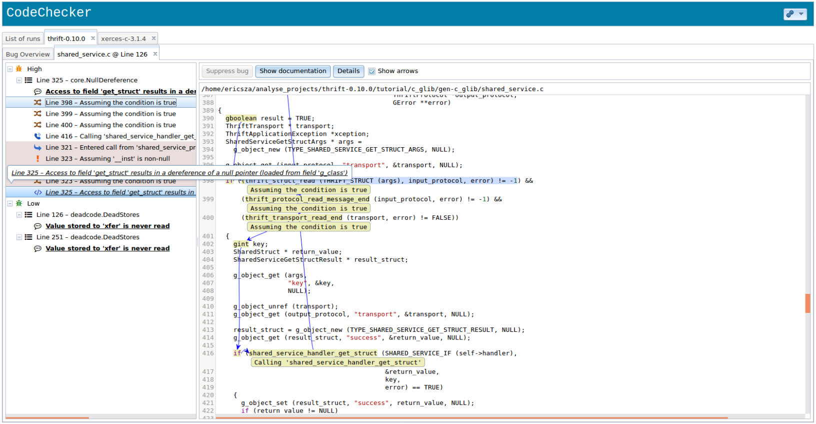
Using it in the Real World

There are several ways to run it on a large project.

- Scan-build-py. Included in the repository.
- CodeChecker: https://github.com/Ericsson/codechecker

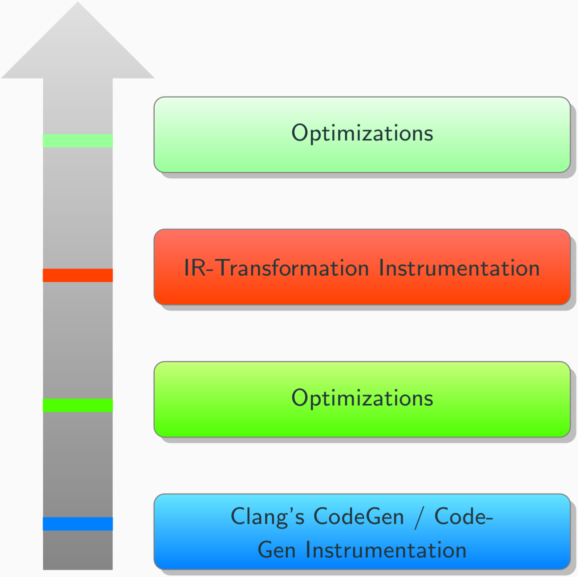
```
scan-build <your build command>
```

CodeChecker



Instrumentation and More

Instrumentation



Instrumentation

In general, you'll want to put the instrumentation at the last place where the necessary semantic information is available.

- If the IR cannot represent the necessary semantic information, you'll need to instrument during Clang's CodeGen.
- Sometimes you can add metadata to the IR during Clang's CodeGen and then insert instrumentation later. Optimizations, however, might drop metadata.
- Running optimizations after instrumentation is inserted can reduce the runtime overhead.
- Running optimizations before instrumentation is inserted can reduce the amount of instrumentation needed (thus reducing the runtime overhead). This can be very important in practice, but sometimes optimizations can eliminate operations that would otherwise be checked, and this is unacceptable.

Instrumentation

Let's consider what UBSan, the undefined-behavior sanitizer, does to our example program:

```
int main(int argc, char *argv[]) {
  return argc-1;
}
```

Compiling so that we can see the IR:

```
$ clang++ -mllvm -disable-llvm-optzns -O3 -S -emit-llvm -o
      - test.cpp
```

- Note the magic way to get the pre-optimization IR while still having Clang generate IR intended for optimization (e.g. including optimization-related metadata). Justing using -O0 is not the same.

Instrumentation

Let's consider what UBSan, the undefined-behavior sanitizer, does to our example program:

```
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  store i32 0, i32* %retval, align 4
  store i32 %argc, i32* %argc.addr, align 4, !tbaa !1
  store i8** %argv, i8*** %argv.addr, align 8, !tbaa !5
  %0 = load i32, i32* %argc.addr, align 4, !tbaa !1
  %sub = sub nsw i32 %0, 1
  ret i32 %sub
}
```

A few things to note about the IR:

- Before optimization, all local variables have stack locations (strictly following the C/C++ abstract-machine model).
- Local stack variables corresponding to function parameters and return values are initialized.
- Here's the actual subtraction.

Instrumentation

Now let's compile with the undefined-behavior (UB) sanitizer by adding `-fsanitize=undefined` to the command line:

```
...
define i32 @main(i32 %argc, i8** nocapture readnone %argv) local_unnamed_addr
       #0 prologue <{ i32, i8* }> <{ i32 1413876459, i8* bitcast ({ i8*, i8*
       }*
 @_ZTIFiiPPcE to i8*) }> {
entry:
  %0 = tail call { i32, i1 } @llvm.ssub.with.overflow.i32(i32 %argc, i32 1)
  %1 = extractvalue { i32, i1 } %0, 0
  %2 = extractvalue { i32, i1 } %0, 1
  br i1 %2, label %handler.sub_overflow, label %cont, !prof !1, !nosanitize
        !2

handler.sub_overflow:
  %3 = zext i32 %argc to i64, !nosanitize !2
  tail call void @__ubsan_handle_sub_overflow(i8* bitcast ({ { [9 x i8]*, i32
        , i32 }, { i16, i16, [6 x i8] }* }* @1 to i8*), i64 %3, i64 1) #3, !
        nosanitize !2
  br label %cont, !nosanitize !2

cont:
  ret i32 %1
...
```

Instrumentation

Where does this come from? Look in Clang's `lib/CodeGen/CGExprScalar.cpp` and we'll find:

```
Value *ScalarExprEmitter::EmitSub(const BinOpInfo &op) {
  ...
  if (op.Ty->isUnsignedIntegerType() &&
      CGF.SanOpts.has(SanitizerKind::UnsignedIntegerOverflow) &&
      !CanElideOverflowCheck(CGF.getContext(), op))
    return EmitOverflowCheckedBinOp(op);

  ...
  return Builder.CreateSub(op.LHS, op.RHS, "sub");
}
```

You'll also likely find the implementation of EmitOverflowCheckedBinOp informative. You'll probably want to specifically note the implementation of CodeGenFunction::EmitCheckTypeDescriptor in CGExpr.cpp.

IR Customization

How to we get “extra” information into the IR? We can use intrinsics, attributes, and metadata.

Intrinsics are internal functions with semantics defined directly by LLVM. LLVM has both target-independent and target-specific intrinsics.

```
define void @test6(i8 *%P) {
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %P, i8* %P, i64 8, i32 4, i1 false
       )
  ret void
}
```

LLVM itself defines the meaning of this call (and the MemCpyOpt transformation will remove this one because it has no effect).

IR Customization

Attributes: Properties of functions, function parameters, or function return values that are part of the function definition and/or callsite itself.

```
define noalias i32* @foo(%struct.x* byval %a) nounwind {
  ret i32* undef
}
```

- Function attribute (these can be from the set of pre-defined attributes or arbitrary key/value pairs)
- Function-argument attribute
- Return-value attribute

IR Customization

In the IR, attributes can be grouped into common sets to avoid repetition.

```
define float @f(float %xf) #0 {
  ...
}

attributes #0 = { norecurse nounwind readnone "disable-tail-calls"="false" "
      less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-
      pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="
      false" "stack-protector-buffer-size"="8" "target-cpu"="pwr8" "target-
      features"="+altivec,+bpermd,+crypto,+direct-move,+extdiv,+power8-
      vector,+vsx,-qpx" "unsafe-fp-math"="false" "use-soft-float"="false" }
```

IR Customization

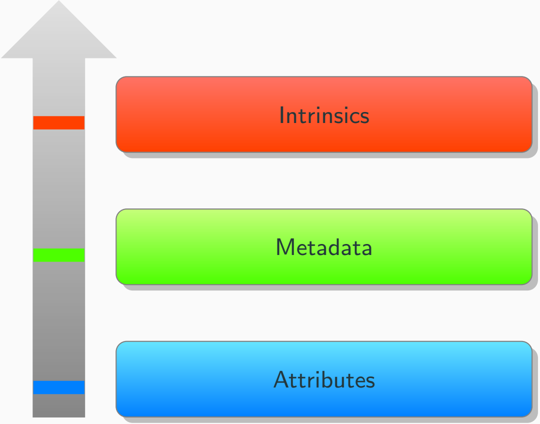
Metadata represents optional information about an instruction (or module) that can be discarded without affecting correctness.

```
define zeroext i1 @_Z3fooPb(i8* nocapture %x) {
entry:
  %a = load i8* %x, align 1, !range !0
  %b = and i8 %a, 1
  %tobool = icmp ne i8 %b, 0
  ret i1 %tobool
}

!0 = !{i8 0, i8 2}
```

- Range metadata provides the optimizer with additional information on a loaded value. %a here is 0 or 1.

IR Customization



- Attrbitues: essentially free, use whenever you can
- Metadata: processing lots of metadata can slow down the optimizer
- Intrinsics: extra instructions and value uses which can also inhibit transformations!

IR Customization

We now also have operand bundles, which are like metadata for calls, but it is illegal to drop them...

```
call void @y() [ "deopt"(i32 10), "unknown"(i8* null) ]
```

- For example, used for implementing deoptimization.
- These are essentially free, like attributes, but can block certain optimizations!

assume Intrinsic

@llvm.assume can provide the optimizer with additional control-flow-dependent truths: Powerful but use sparingly!

Why sparingly? Additional uses are added to variables you care about optimizing, and that can block optimizations. But sometimes you care more about the information being added than these optimizations: pointer alignments are a good example.

```
define i32 @foo1(i32* %a) {
entry:
  %0 = load i32* %a,  align 4

  %ptrint = ptrtoint i32* %a to i64
  %maskedptr = and i64 %ptrint, 31
  %maskcond = icmp eq i64 %maskedptr, 0
  tail call void @llvm.assume(i1 %maskcond)

  ret i32 %0
}
```

- InstCombine will make this align 32, and the assume call will stay!

assume Intrinsic

```
define i32 @foo1(i32* %a) {
entry:
  %0 = load i32* %a, align 4

  %ptrint = ptrtoint i32* %a to i64
  %maskedptr = and i64 %ptrint, 31
  %maskcond = icmp eq i64 %maskedptr, 0
  tail call void @llvm.assume(i1 %maskcond)

  ret i32 %0
}
```

- But what about all of these extra values? Don't they affect loop unrolling, inlining, etc.? These are: Ephemeral values!

Ephemeral values are collected by collectEphemeralValues, a utility function in CodeMetrics, and excluded from the cost heuristics used by the inliner, loop unroller, etc.

The AssumptionTracker

Assumptions are control-flow dependent, how can you find the relevant ones?

A new module-level "invariant" analysis, the AssumptionTracker, keeps track of all of the assumes currently in the module. So finding them is easy:

```
for (auto &AssumeCall : AT->assumptionsFor(V)) {
  ...
}
```

If you create a new assume, you'll need to register it with the AssumptionTracker:

```
AT->registerAssumption(CI);
```

Also, note the isValidAssumeForContext function from ValueTracking.

Loop Metadata

Fundamental question: How can you attach metadata to a loop?

LLVM has no fundamental IR construction to represent a loop, and so the metadata must be attached to some instruction; which one?

```
  br i1 %exitcond, label %._crit_edge, label %.lr.ph, !llvm.loop !0
...
!0 = distinct !{ !0, !1 }
!1 = !{ !"llvm.loop.unroll.count", i32 4 }
```

- The backedge branch gets the metadata
- It is important to mark your loop IDs as "distinct" metadata

Loop Metadata

Fundamental question: How can you identify the source location of a loop?

- The source location of a loop is needed when generating optimization remarks regarding loops.
- We used to guess based on the source location of the preheader's branch (if it exists) or the branch of the loop header.

Now we have a better way: Put the debug-info source location in the metadata loop id:

```
  br i1 %exitcond, label %._crit_edge, label %.lr.ph, !llvm.loop !0
...
!0 = distinct !{ !0, !1, !2 }
!1 = !{ !"llvm.loop.unroll.count", i32 4 }
!2 = !DILocation(line: 2, column: 3, scope: ...)
```

Fast-Math Flags

How do we model -ffast-math and friends?

- `%op1 = fadd nnan float %load1, 1.0` - Assume that the arguments and result are not NaN.
- `%op1 = fadd ninf float %load1, 1.0` - Assume that the arguments and result are not +/-Inf.
- `%op1 = fadd nsz float %load1, 1.0` - Assume that the sign of zero is insignificant.
- `%op1 = fadd arcp float %load1, 1.0` - We can use the reciprocal rather than perform division.
- `%op1 = fadd fast float %load1, 1.0` - Allow algebraically-equivalent transformations (implies all others)

Metadata / Attribute APIs

Some useful APIs:

```
...
auto *MD = I->getMetadata("nosanitize");
...
MDNode *LoopID = L->getLoopID();
...
StringRef TrapFuncName =
    I.getAttributes()
        .getAttribute(AttributeSet::FunctionIndex, "trap-func-name")
        .getValueAsString();
...
Attribute FSAttr = F.getFnAttribute("target-features");
...
```

Prefix/Prologue Data

The problem: How to embed runtime-accessible metadata with functions (that you can get if you have the function pointer)...

```
define void @f() prefix i32 123 { ... }
```

The problem: How to embed arbitrary data (likely code) at the beginning of a function...

```
define void @f() prologue i8 144 { ... }
```

- Here's the data: An i32 with the value: 123
- On x86, 144 is a nop.

Alias Analysis

```
...
if (!AA.isNoAlias(MemoryLocation::get(SI), LoadLoc))
...
```

where MemoryLocation has a pointer value, a size (which might be unknown), and optionally, AA metadata from the associated access.

You can directly examine the AA results for some given IR like this:

```
$ opt -basicaa -aa-eval -print-all-alias-modref-info -
    disable-output < input.ll
```

There are several kinds of aliasing results:

- NoAlias - The two locations do not alias at all.
- MayAlias - The two locations may or may not alias. This is the least precise result.
- PartialAlias - The two locations alias, but only due to a partial overlap.
- MustAlias - The two locations precisely alias each other.

Alias Analysis

In many cases, what you want to know is: Does a given memory access (e.g. load, store, function call) alias with another memory access? For this kind of query, we have the Mod/Ref interface...

```
...
if (AA.getModRefInfo(&*I, StoreLoc) != MRI_NoModRef)
...
```

where we have:

```
enum ModRefInfo {
  /// The access neither references nor modifies the value stored in memory.
  MRI_NoModRef = 0,
  /// The access references the value stored in memory.
  MRI_Ref = 1,
  /// The access modifies the value stored in memory.
  MRI_Mod = 2,
  /// The access both references and modifies the value stored in memory.
  MRI_ModRef = MRI_Ref | MRI_Mod
};
```

Also useful, we have an AliasSetTracker to collect aliasing pointers into disjoint sets.

Alias Analysis

A note on loops and the meaning of aliasing; consider...

```
void foo(double *a) {
  for (int i = 0; i < 1600; ++i)
    a[i+1] = a[i] + 1.0;
}
```

where we might represent the loop in IR as:

```
for.body:                                          ; preds = %for.body , %entry
  %indvars.iv = phi i64 [ 0, %entry ], [ %indvars.iv.next , %for.body ]
  %arrayidx = getelementptr inbounds double, double* %a, i64 %indvars.iv
  %0 = load double, double* %arrayidx , align 8
  %add = fadd double %0, 1.000000e+00
  %indvars.iv.next = add nuw nsw i64 %indvars.iv , 1
  %arrayidx2 = getelementptr inbounds double, double* %a, i64 %
      indvars.iv.next
  store double %add, double* %arrayidx2 , align 8
  %exitcond = icmp eq i64 %indvars.iv.next , 1600
  br i1 %exitcond , label %for.cond.cleanup , label %for.body
```

- Do the load and store alias? No. Aliasing is defined only at a potential point in the control flow, not over all values a phi might hold.

Alias Analysis

Given some memory access, how do you know on what memory accesses it depends? Use our MemorySSA analysis. For example, running:

```
opt -basicaa -print-memoryssa -verify-memoryssa -
    analyze < input.ll
```

```
define i32 @main() {
entry:
; CHECK: 1 = MemoryDef(liveOnEntry)
  %call = call noalias i8* @_Znwm(i64 4)
  %0 = bitcast i8* %call to i32*
; CHECK: 2 = MemoryDef(1)
  %call1 = call noalias i8* @_Znwm(i64 4)
  %1 = bitcast i8* %call1 to i32*
; CHECK: 3 = MemoryDef(2)
  store i32 5, i32* %0, align 4
; CHECK: 4 = MemoryDef(3)
  store i32 7, i32* %1, align 4
; CHECK: MemoryUse(3)
  %2 = load i32, i32* %0, align 4
; CHECK: MemoryUse(4)
  %3 = load i32, i32* %1, align 4
; CHECK: MemoryUse(3)
  %4 = load i32, i32* %0, align 4
; CHECK: MemoryUse(4)
  %5 = load i32, i32* %1, align 4
  %add = add nsw i32 %3, %5
  ret i32 %add
}

declare noalias i8* @_Znwm(i64)
```

Alias Analysis

What kinds of alias analysis is in LLVM?

```
  AAR->addAAResult(getAnalysis<BasicAAWrapperPass>().getResult());

// Populate the results with the currently available AAs.
if (auto *WrapperPass = getAnalysisIfAvailable<ScopedNoAliasAAWrapperPass
      >())
  AAR->addAAResult(WrapperPass->getResult());
if (auto *WrapperPass = getAnalysisIfAvailable<TypeBasedAAWrapperPass>())
  AAR->addAAResult(WrapperPass->getResult());
if (auto *WrapperPass =
        getAnalysisIfAvailable<objcarc::ObjCARCAAWrapperPass>())
  AAR->addAAResult(WrapperPass->getResult());
if (auto *WrapperPass = getAnalysisIfAvailable<GlobalsAAWrapperPass>())
  AAR->addAAResult(WrapperPass->getResult());
if (auto *WrapperPass = getAnalysisIfAvailable<SCEVAAWrapperPass>())
  AAR->addAAResult(WrapperPass->getResult());
if (auto *WrapperPass = getAnalysisIfAvailable<CFLAndersAAWrapperPass>())
  AAR->addAAResult(WrapperPass->getResult());
if (auto *WrapperPass = getAnalysisIfAvailable<CFLSteensAAWrapperPass>())
  AAR->addAAResult(WrapperPass->getResult());

// If available, run an external AA providing callback over the results as
// well.
if (auto *WrapperPass = getAnalysisIfAvailable<ExternalAAWrapperPass>())
  if (WrapperPass->CB)
    WrapperPass->CB(*this, F, *AAR);
```

Optimization Reporting

Goal: To get information from the backend (LLVM) to the frontend (Clang, etc.)

- To enable the backend to generate diagnostics and informational messages for display to users.
- To enable these messages to carry additional "metadata" for use by knowledgeable frontends/tools
- To enable the programmatic use of these messages by tools (auto-tuners, etc.)
- To enable plugins to generate their own unique messages

```
sqlite3.c:60198:7: remark: sqlite3StrICmp inlined into sqlite3Pragma [-Rpass=inline]
  if( sqlite3StrICmp(zLeft, "case_sensitive_like")==0 ){
      ^
sqlite3.c:60200:40: remark: getBoolean inlined into sqlite3Pragma [-Rpass=inline]
      sqlite3RegisterLikeFunctions(db, getBoolean(zRight));
                                       ^
sqlite3.c:60213:7: remark: sqlite3StrICmp inlined into sqlite3Pragma [-Rpass=inline]
  if( sqlite3StrICmp(zLeft, "integrity_check")==0
      ^
sqlite3.c:60214:7: remark: sqlite3StrICmp inlined into sqlite3Pragma [-Rpass=inline]
   || sqlite3StrICmp(zLeft, "quick_check")==0
      ^
sqlite3.c:44776:8: remark: sqlite3VdbeMemFinalize inlined into sqlite3VdbeExec [-Rpass=inline]
  rc = sqlite3VdbeMemFinalize(pMem, pOp->p4.pFunc);
       ^
```

Optimization Reporting

There are two ways to get the diagnotics out:

- By using some fontend that installs the relevant callbacks and displays the associated messages
- By serializing the messages to a file (YAML is currently used as the format), and then processing them with an external tool

```
$ clang -O3 -c -o /tmp/or.o /tmp/or.c -fsave-
    optimization-record
$ llvm-opt-report /tmp/or.opt.yaml
```

```
< /tmp/or.c
 1          | void bar();
 2          | void foo() { bar(); }
 3          |
 4          | void Test(int *res, int *c, int *d, int *p, int n) {
 5          |   int i;
 6          |
 7          | #pragma clang loop vectorize(assume_safety)
 8     V4,2 |   for (i = 0; i < 1600; i++) {
 9          |     res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
10          |   }
11          |
12  U16     |   for (i = 0; i < 16; i++) {
13          |     res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
14          |   }
15          |
16 I        |   foo();
17          |
18          |   foo(); bar(); foo();
   I        |   ^
   I        |                 ^
19          | }
20          |
```

Optimization Reporting

But how code is optimized often depends on where it is inlined...

```
< /tmp/q.cpp
 1     | void bar();
 2     | void foo(int n) {
 [[
  > foo(int):
 3     |   for (int i = 0; i < n; ++i)
  > quack(), quack2():
 3  U4 |   for (int i = 0; i < n; ++i)
 ]]
 4     |     bar();
 5     | }
 6     |
 7     | void quack() {
 8 I   |   foo(4);
 9     | }
10     |
11     | void quack2() {
12 I   |   foo(4);
13     | }
14     |
```

A viewer that creates HTML reports is also under development (the current prototype, called opt-viewer, is in LLVM's utils directory).

Optimization Reporting

What's in these YAML files?


```yaml
--- !Missed
Pass:            inline
Name:            NoDefinition
DebugLoc:        { File: Inputs/q3.c, Line: 4, Column: 5 }
Function:        foo
Args:
  - Callee:           bar
  - String:           ' will not be inlined into '
  - Caller:           foo
  - String:           ' because its definition is unavailable'
...
--- !Analysis
Pass:            inline
Name:            CanBeInlined
DebugLoc:        { File: Inputs/q3.c, Line: 8, Column: 3 }
Function:        quack
Args:
  - Callee:           foo
  - String:           ' can be inlined into '
  - Caller:           quack
  - String:           ' with cost='
  - Cost:             '40'
  - String:           ' (threshold='
  - Threshold:        '275'
  - String:           ')'
...
```

Optimization Reporting

How can a frontend receive callbacks or activate optimization recording?

See Clang's lib/CodeGen/CodeGenAction.cpp:

```
...
  LLVMContext::DiagnosticHandlerTy OldDiagnosticHandler =
     Ctx.getDiagnosticHandler();
  void *OldDiagnosticContext = Ctx.getDiagnosticContext();
  Ctx.setDiagnosticHandler(DiagnosticHandler, this);
  Ctx.setDiagnosticHotnessRequested(CodeGenOpts.DiagnosticsWithHotness);

  std::unique_ptr<llvm::tool_output_file> OptRecordFile;
  if (!CodeGenOpts.OptRecordFile.empty()) {
  std::error_code EC;
  OptRecordFile =
     llvm::make_unique<llvm::tool_output_file>(CodeGenOpts.OptRecordFile,
         EC, sys::fs::F_None);
  if (EC) {
     ...
  }

  Ctx.setDiagnosticsOutputFile(
        llvm::make_unique<yaml::Output>(OptRecordFile->os()));
...
```

Optimization Reporting

Emitting remarks from optimization passes is also fairly easy. For examples, look at the inliner and loop vectorizer.

```
...
    ORE.emit(OptimizationRemarkAnalysis(DEBUG_TYPE, "TooCostly", Call)
             << NV("Callee", Callee) << " too costly to inline (cost="
             << NV("Cost", IC.getCost()) << ", threshold="
             << NV("Threshold", IC.getCostDelta() + IC.getCost()) << ")");
...
    ORE.emit(OptimizationRemarkAnalysis(DEBUG_TYPE, "NeverInline", Call)
             << NV("Callee", Callee)
             << " should never be inlined (cost=never)");
...
```

Note the use of the "NV" type, a name/value pair.

Address Sanitizer

Address Sanitizer detects things like:

- Buffer overruns (and underruns)
- Use after free and duplicate frees

How does it work?

- Instrument all loads and stores
- Add “red zones” around stack variables and globals
- Hook memcpy, etc.
- Provide a memory allocator which tracks state and has allocation “red zones”

Address Sanitizer

Address Sanitizer uses shadow memory; for each shadow byte:

- 0 means that all 8 bytes of the corresponding application memory region are addressable
- $k (1 \leq k \leq 7)$ means that the first k bytes are addressable
- any negative value indicates that the entire 8-byte word is unaddressable (different kinds of unaddressable memory (heap redzones, stack redzones, global redzones, freed memory) use different negative values)

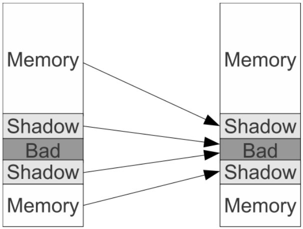


Figure 1: AddressSanitizer memory mapping.

Address Sanitizer

Address Sanitizer is in compiler-rt in lib/asan. A lot of the common "sanitizer" infrastructure is in lib/sanitizer_common. This provides infrastructure you can use for your own tools:

```
class Decorator: public __sanitizer::SanitizerCommonDecorator {
 public:
  Decorator() : SanitizerCommonDecorator() { }
  const char *Warning()    { return Red(); }
  const char *End()    { return Default(); }
};
  ...
  Decorator d;
  Printf("%s", d.Warning());
  Report("ERROR: Something bad on address %p (tid %d)\n",
         Addr, GetTid());
  Printf("%s", d.End());
  ...
  BufferedStackTrace ST;
  ST.Unwind(kStackTraceMax, pc, bp, 0, 0, 0, false);
  ST.Print();
  ...
INTERCEPTOR(void *, memset, void *dst, int v, uptr size) {
  if (REAL(memset) == nullptr)
    return internal_memset(dst, v, size);
  void *res = REAL(memset)(dst, v, size);
  ...
  return res;
}
```

PGO

Briefly, for instrumentation-based profiling:

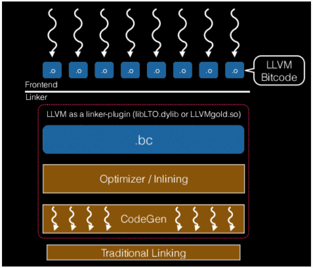
```
$ clang++ -O2 -fprofile-instr-generate code.cc -o code
$ LLVM_PROFILE_FILE="code-%p.profraw" ./code
$ llvm-profdata merge -output=code.profdata code-*.
    profraw
$ clang++ -O2 -fprofile-instr-use=code.profdata code.cc
     -o code
```

For more information, see https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization.

```
...
const BranchProbability ColdProb(1, 5); // 20%
ColdEntryFreq = BlockFrequency(BFI->getEntryFreq()) * ColdProb;
...
BlockFrequency LoopEntryFreq = BFI->getBlockFreq(L->getLoopPreheader());
...
```

LTO

If you need to see all of the code at once, you can use link-time optimization (use the flag `-flto when compiling and linking`):



For more information, see `http://llvm.org/devmtg/2016-11/Slides/Amini-Johnson-ThinLTO.pdf`.

Vectorization

We have two production-quality vectorizers in LLVM: The loop vectorizer (which currently only vectorizes inner loops) and the SLP (superword-level parallelism) vectorizer. These produce vectorized IR using LLVM's vector types:

```
...
vector.body:                                      ; preds = %vector.body, %
      entry
  %index = phi i64 [ 0, %entry ], [ %index.next, %vector.body ]
  %0 = getelementptr inbounds double, double* %b, i64 %index
  %1 = bitcast double* %0 to <4 x double>*
  %wide.load = load <4 x double>, <4 x double>* %1, align 8, !tbaa !1
  %2 = fadd <4 x double> %wide.load, <double 1.000000e+00, double 1.000000e
        +00, double 1.000000e+00, double 1.000000e+00>
  %3 = getelementptr inbounds double, double* %a, i64 %index
  %4 = bitcast double* %3 to <4 x double>*
  store <4 x double> %2, <4 x double>* %4, align 8, !tbaa !1
  %index.next = add i64 %index, 4
  %5 = icmp eq i64 %index.next, 1600
  br i1 %5, label %for.cond.cleanup, label %vector.body, !llvm.loop !5
...
```

Vectorization

How do the vectorizers know what makes sense for the current target? They use TargetTransformInfo (TTI), see include/llvm/Analysis/TargetTransformInfo.h:

```
...
unsigned getNumberOfRegisters(bool Vector) const;
unsigned getRegisterBitWidth(bool Vector) const;
...
int getArithmeticInstrCost(
    unsigned Opcode, Type *Ty, OperandValueKind Opd1Info = OK_AnyValue,
    OperandValueKind Opd2Info = OK_AnyValue,
    OperandValueProperties Opd1PropInfo = OP_None,
    OperandValueProperties Opd2PropInfo = OP_None,
    ArrayRef<const Value *> Args = ArrayRef<const Value *>()) const;
...
int getShuffleCost(ShuffleKind Kind, Type *Tp, int Index = 0,
                   Type *SubTp = nullptr) const;
...
int getCastInstrCost(unsigned Opcode, Type *Dst, Type *Src) const;
...
```

Note that TTI contains two cost models: One set of functions used by the vectorizers which primarily deal in reciprocal throughputs and one set (getUserCost and friends) used by the inliner and loop unrollers.