Mini Assignment 2

Akilesh B, CS13B1042 and Phaneendra Babu, CS13B1037

October 20, 2015

Options provided to view the LLVM/Clang AST

- There are options to control error and warning messages

  -Werror : Turn warnings into errors.

  -Werror=foo : Turn warning “foo” into an error.

  -Wno-error=foo : Turn warning “foo” into an warning even if -Werror is specified.

  -Wfoo : Enable warning “foo”.

  -Wno-foo : Disable warning “foo”.

  -w : Disable all diagnostics.

  -Weverything : Enable all diagnostics.

  -pedantic : Warn on language extensions.

  -pedantic-errors : Error on language extensions.

  -Wsystem-headers : Enable warnings from system headers.

- Clang provides a wide range of options to control the exact output format of the diagnostics that it generates.

  -f[no-]show-column : Print column number in diagnostic.

  -f[no-]show-source-location : Print source file/line/column information in diagnostic.

  -f[no-]caret-diagnostics : Print source line and ranges from source code in diagnostic.

  -f[no-]diagnostics-show-option : Enable [-Woption] information in diagnostic line.

- Clang does crash from time to time. Clang generates preprocessed source file(s) and associated run script(s) upon a crash.

  -fno-crash-diagnostics : Disable auto-generation of preprocessed source files during a clang crash.

- Options to Emit Optimization Reports

  Optimization reports trace, at a high-level, all the major decisions done by compiler transformations. For instance, when the inliner decides to inline function foo() into bar(), or the loop unroller decides to unroll a loop N times, or the vectorizer decides to vectorize a loop body.

AST Structure (Design of LLVM-AST)

- "clang -cc1 -ast-dump your_file.c/.cpp" is used to view the AST.

- All information about the AST for a translation unit is bundled up in the class ASTContext.

- It allows traversal of the whole translation unit starting from TranslationUnitDecl.
- Clang's AST nodes are modeled on a class hierarchy that does not have a common ancestor.
- In our previous ANTLR parser assignment there are some common ancestor classes such as expression, Object.
- Instead, there are multiple larger hierarchies for basic node types like Decl and Stmt.
- Many important AST nodes derive from Type, Decl, DeclContext or Stmt, with some classes deriving from both Decl and DeclContext.
- Thus, to traverse the full AST, one starts from the TranslationUnitDecl and then recursively traverses everything that can be reached from that node - this information has to be encoded for each specific node type.
- The two most basic nodes in the Clang AST are statements (Stmt) and declarations (Decl). expressions (Expr) are also statements in Clang's AST.

AST Traversal

- FrontEnd of LLVM uses RecursiveASTVisitor to find CXXRecordDecl AST nodes with a specified name.
- The common entry point to produce syntax tree is the FrontendAction.
- FrontendAction is an interface that allows execution of user specific actions as part of the compilation.
- To run tools over the AST clang provides the convenience interface ASTFrontendAction, which takes care of executing the action.
- CreateASTConsumer returns ASTConsumer per translation unit.
- ASTConsumer is an interface used to write generic actions on an AST, regardless of how the AST was produced.
- ASTConsumer provides many different entry points, but for our use case the only one needed is HandleTranslationUnit, which is called with the ASTContext for the translation unit.
- Traversing the translation unit decl via a RecursiveASTVisitor will visit all nodes in the AST.
- RecursiveASTVisitor is written to visit all CXXRecordDecl's
- For debugging, dumping the AST nodes will show which nodes are already being visited.

- In the methods of our RecursiveASTVisitor we can now use the full power of the Clang AST to drill through to the parts that are interesting for us. For example, to find all class declaration with a certain name, we can check for a specific qualified name.
- Some of the information about the AST, like source locations and global identifier information, are not stored in the AST nodes themselves, but in the ASTContext and its associated source manager.
- To retrieve them we need to hand the ASTContext into our RecursiveASTVisitor implementation. Now that the ASTContext is available in the RecursiveASTVisitor. We can also look into source locations.

Error messages

- ErrorHandling.h file is studied for this report
- install_fatal_error_handler function Installs a new error handler to be used whenever a serious (non-recoverable) error is encountered by LLVM.
- If no error handler is installed the default is to print the error message to stderr, and call exit(1).
- If an error handler is installed then it is the handler's responsibility to log the message, it will no longer be printed to stderr. If the error handler returns, then exit(1) will be called.
- It is dangerous to naively use an error handler which throws an exception. Even though some applications desire to gracefully recover from arbitrary faults, blindly throwing exceptions through unfamiliar code isn't a way to achieve this.
- remove_fatal_error_handler() : restores default error handling behaviour
- ScopedFatalErrorHandler - This is a simple helper class which just calls install_fatal_error_handler in its constructor and remove_fatal_error_handler in its destructor.
- report_fatal_error(...) : Reports a serious error, calling any installed error handler. These functions are intended to be used for error conditions which are outside the control of the compiler (I/O errors, invalid user input, etc.)
- llvm_unreachable_internal(...) : This function calls abort(), and prints the optional message to stderr. Use the llvm_unreachable macro (that adds location info), instead of calling this function directly.
- llvm_unreachable(...) : Marks that the current location is not supposed to be reachable.In !NDEBUG builds, prints the message and location info to stderr. In NDEBUG builds, becomes an optimizer hint that the current

location is not supposed to be reachable. On compilers that don't support such hints, prints a reduced message instead.

Assert mechanism

- The analyzer exploits code assertions by pruning off paths where the assertion condition is false.
- The idea is capture any program invariants specified in the assertion that the developer may know but is not immediately apparent in the code itself.
- In this way assertions make implicit assumptions explicit in the code, which not only makes the analyzer more accurate when finding bugs, but can help others better able to understand your code as well. It can also help remove certain kinds of analyzer false positives by pruning off false paths.
- In order to exploit assertions, however, the analyzer must understand when it encounters an "assertion handler."
- Typically assertions are implemented with a macro, with the macro performing a check for the assertion condition and when the check fails, calling an assertion handler.

Specific error messages

- Unknown Type Names : One annoying thing about parsing C and C++ is that you have to know what is a typename in order to parse the code. error: unknown type name
- Spell Checker : One of the more visible things that Clang includes is a spell checker (also on reddit). The spell checker kicks in when we use an identifier that Clang doesn't know: it checks against other close identifiers and suggests what you probably meant.
- Clang tracks the typedefs you write in your code carefully so that it can relate errors to the types you use in your code. This allows it to print out error messages in your terms, not in fully resolved and template instantiated compiler terms
- It also uses its range information and caret to show you what you wrote instead of trying to print it back out at you.
- One mistake many beginner programmers mistake is that they accidentally define functions instead of objects on the stack. This is due to an ambiguity in the C++ grammar which is resolved in an arbitrary way. This is an unavoidable part of C++, compiler should help you underatsnd what's going wrong.
- Missing semicolons is very frequent mistake.

LLVM-IR and assembly code:

- To generate IR “clang -Os -S -emit-llvm test.c -o test.ll”
- IR is a low level language that lies between high-level program and low-level backend.
- It is strongly typed with a simple type system. For eg. i32 is a 32-bit integer.
- It is the core of LLVM. Front end compiles the code from a source language to IR and code generators turn the IR into machine code.
- The generated IR begins with ModuleID (name of C program), target datalayout and target triple (which contain details about architecture and operating system of target machine).
- Any printf in C code is converted into string constant @.str This has some qualifiers like internal, constant, unnamed_addr. The square bracket which follows indicates that it’s an array.
- Any function present in C code is present as ; Function Attrs: in IR code The function has attributes like: nounwind, optsize, uwtable.
- nounwind indicates that the function that doesn’t throw any exception and can be used for optimization later.
- uwtable - indicates that Application binary interface (ABI) being targeted requires an unwind table entry produced for this function.
- Every function consists of an entry section, return section (in case it returns some value).
- The main function doesn’t contain any branches, it’s a single block. Basic blocks are terminated with some flow-control instruction.
- The type returned by the ret instruction and the return type specified in function definition must match. If not, the IR will fail to validate.
- for loops are present within: for.cond, for.body and for.end. If condition is present as: if.then section and if.end section.
- The calling convention is abstracted through call and ret instructions and explicit arguments. It doesn’t use a fixed set of named registers, it uses an infinite set of temporaries named with a % character.
- LLVM views global variables as pointers, so we must explicitly dereference the global variable using the "load" instruction when accessing its value.
- Two kinds of local variables: i) Register-allocated local variables or temporaries, ii) Stack-allocated local variables.

- There are 9 types of cast operators: 1) Zero-extending casts (unsigned casts), 2) Bitwise casts (type casts), 3) Sign-extending casts (signed upcasts), 4) Truncating casts (signed and unsigned downcasts), 5) Floating-point extending casts (float upcasts), 6) Pointer to integer casts, 7) Integer to pointer casts, 8) Address-space casts, 9) Floating-point truncating casts (float downcasts).

General observations:

- Comments in LLVM assembly begin with a semicolon (;) and go on till the end of the line.
- Global identifiers begin with the at (@) character. All function names and global variables must begin with @, as well.
- Local identifiers in the LLVM begin with a percent symbol (%).
- The common regular expression for identifiers is [%@][a-zA-Z$._][a-zA-Z$._0-9]*.
- One important feature of LLVM is its strong type system. The LLVM defines an integer type as iN, where N is the number of bits the integer will occupy. You can specify any bit width between 1 and 223- 1.
- A vector or array type can be declared as [no. of elements X size of each element]. For instance, the string "Hello" this makes the type [6 x i8], assuming that each character is 1 byte and 1 extra byte for the NULL character.
- A global string constant for the hello string can be declared as follows: @hello = constant [6 x i8] c "Hello\00".
- Use the constant keyword to declare a constant followed by the type and the value.
- The type begins by using c followed by the entire string in double quotation marks, including \0 and ending with 0. There is no mention of why string needs to be declared with the c prefix and include both a NULL chatacter and 0 at the end in the LLVM documentation.
- Function begins with the define keyword followed by the return type, and then the function name. A simple definition of main that returns a 32-bit integer similar to: *define i32 @main() { ; some LLVM assembly code which returns i32 }*.
- Function declaration of a simple puts method, which is the LLVM equivalent of printf: *declare i32 puts(i8*)*. You begin the declaration with the declare keyword followed by the return type, the function name, and an optional list of arguments to the function. The declaration must be in the global scope.

- Each function ends with a return statement. There are two forms of return statement: *ret <type> <value> or ret void.*
- Functions can be called as follows: *call <function return type> <function name> <optional function arguments>.* Note that each function argument must be preceded by its type. A function test that returns an integer of 8 bits and accepts an integer of 46 bits has the syntax: *call i8 @test(i46 %arg1).*
- Whenever we have overloaded functions in C++, llvm will encode type information in the symbol name. This is name mangling and it applied to all symbols even when there is no conflict. It would be an error in C as C doesn't permit two functions with the same name.
- From IR its very easy to compile it to a machine code for any architecture.
- IR is better than assembly in following ways:
  - Assembly code might not use faster instructions available in CPU of future generations
  - Neither can it work on older CPU's
  - Porting assembly to another CPU architecture is tedious.
  - IR has all advantages of assembly and none of its major problems: fast, expressive, retargettable and maintainable.

Misc. observations:

- Source language types are lowered.
  - Rich type systems are expanded to simple type system.
  - Implicit and abstract types are made explicit and concrete.
- Examples:
  - References turn into pointers: T& -> T*
  - Complex numbers: complex float -> {float, float}
  - Inheritance: class T : S { int X; } -> {S, int}
  - Methods: class T { void foo(); } -> void foo(T*)
- Low-level and target independent semantics:
  - It has infinite virtual register set in SSA form.
  - Low-level control flow constructs.
  - Load/Store instructions with typed-pointers.
- High-level information

  - Explicit control-flow graph.
  - Explicit data-flow through SSA form.
  - Explicit language independent type information.
  - Explicitly typed pointer arithmetic which preserves array subscript and structure indexing.

Extending Kaleidoscope

- Support user-defined operator (introduce new operators, change precedence level etc.)
- First we need to add lexer extensions.
  - Add enum values for relevant tokens.
  - Recognize new key words in the lexer.
- AST Extension - Add a new AST node for it which has pointers to subexpressions.
- Define a new parsing function.
- The most important step is adding LLVM IR support.
- For eg: when we want to add binary operators.
- define token: tok_binary = some integer.
- Extend the PrototypeAST node. This is essential in understanding the precedence level of the operator
- Add support for parsing.
- Add codegen support. It will be an addition of default case to the existing binary operator node.
- The code we add should do a lookup for corresponding operator in the symbol table and generate a function call to it. It is because user-defined operators are also ordinary functions.
- Before calling codegen for a function, if we encounter a user-defined operator, we register it in the precedence table.

References:

1) LLVM Compiler Project: llvm.org