GROUP 13
CS3020
19 October 2015

Clang and LLVM architecture

Mini-Programming Assignment #2

1. A BRIEF INTRODUCTION TO CLANG

The Clang Compiler is an open-source compiler for the C family of programming languages; it is based on the LLVM optimiser and code generator and can be used for multiple targets. It can be used in conjunction with several different types of compilers such as GCC and Windows, however its features are still heavily dependent on the computer architecture that it is operating on.

Clang is a product of the LLVM and Apple community, with the objective of creating a better and more user friendly compiler front-end. This front-end involves better diagnostics and command line tools. Unlike GCC it allows the user to specify the exact format of the diagnostics that it generates. Its code is simple and easily extensible by users who wish to tailor-make it for their own purpose. In contrary to GCC, which integrates multiple preprocessors, the work of Clang is based on a unified parser and better performances such as low memory footprint and lazy evaluation.

However, Clang seeks to maintain its uniqueness in terms of being non-pragmatic. It does not implement features such as variable length arrays, in order to stick to the more robust and used features with cleaner and extensible implementations. Also, it is reserved to only the family of C, with no intentions of incorporating languages such as Java or FORTRAN. Having said that, it supports several unique features for each individual language. For instance all standard features for C are supported apart from the

C99 floating-point pragmas. Also, it caters to all the features of C++98 except for the exported templates, which was removed by C++11.

An example, as mentioned earlier, is Clangs ability to modify the format of diagnostics:

`_opt_fshow-source-location:` `controls whether or not Clang`

```
   prints the filename, line number and column number of a
diagnostic in its error message such as:
          test.c:28:8: warning: extra tokens at end of #endif
directive [-Wextra-tokens]
          #endif bad
               ^
```

Optimisations such as controlling Code Generation (allowing users to decide whether or not to permit runtime checks for various forms of undefined or suspicious behaviour) and controlling the size of debug information.

2. A BRIEF INTRODUCTION TO LLVM

It is a mid-level Optimiser and Code Generator which basically generates intermediate machine code, when given a suitable front end (parser and lexer) such as Clang. It seeks to be machine independent and can generate codes for multiple targets such as the X86 family, ARM, and MIPS. It is compatible with GCC and later versions of LLVM (such as LLVM 2.0) use Clang as its new C front-end.

The main uses of LLVM include:

1. **In-memory compiler IR:**

   Its Intermediate Representation is similar to that of C, with recent improvements such as removal of the no operation and redundancy elimination.

2. **On-disk bitcode representation:**

   This is used in Just-In-Time compilers.

3. **Human readable assembly language representation.**

Like Clang, the main advantage with LLVM is its modular design, making it user friendly and extensible. Some of the key features of LLVM include its wide variety of Link Time Optimisations for all global variables and functions and Just-in-Time code generation, as mentioned before.

3. DIRECTORY HIERARCHY OF LLVM AND CLANG

One of the main component of CLANG is its AST. With over 100k lines of code based on only the AST, we can comprehend the intricate nature of the various aspects of the AST. It is also fully type resolved, synonymous to the C++ language that it parses.

The main class of the AST is ASTContext. This class stores information about the AST in the identifier table. Information regarding the entry point is also taken care by the ASTContext, using a feature known as TranslationUnitDecl.

The core classes of the AST include the Decl, Stmt, and Type class. Each of these classes are independent identities, since there is no base class in AST. The main issue with having no base class, is that one cannot traverse the entire tree in continuation. I.e, there is no way to traverse from one class to another.

Core Classes:

1. Decl

1. CXXRecordDecl: structs, *unions and class*

   The CXXRecordDecl is also used to handle Templates, which has an entire AST definition. The types which are unknown are declared as dependent types.

2. VarDecl
3. UnresolvedUsingTypenameDecl

2. Stmt

1. Compound statement
2. CXXTryStmt
3. BinaryOperator: In clang expressions are statements

3. Type

1. PointerType
2. ParentType
3. SubstTemplateTypeParamType

In order to resolve this issue, other classes and methods (known as Glue classes and methods) are used as connecting entities for AST traversals.

Glue Classes:

1. DeclContext

Inherited by decls that contain other decls

2. TemplateArgument

Accessors for the template argument

3. NestedNameSpecifier

4. Qualtype:

```
const int x;
```

Here the QualType “const”, first defines it’s QualType and then points to the actual data type. This is to prevent the enormous numbers of types that could arise, if QualType was incorporated within the core class.

An example of a **Glue method** would be as follows:

```
IfStmt: getThen(), getElse(), getCond().
```

Locations:

These come under the class SourceLocation and are managed by the SourceManager. The main way that they operate is through the getLocStart and the getLocEnd methods, which point to the starting and the ending tokens respectively. All methods are of the form getLoc method such as getQualifierLoc.

Getting the locations of types such as Classes is much easier, since the type can be accessed by a TypeLoc and can be referenced throughout the program.

4. AST STRUCTURE OF LLVM/CLANG

The non-trivial program and the corresponding AST tree is present in the folder for the assignment. Some interesting features of the AST tree are as follows:

4.1.Every header that needs to be preprocessed becomes a part of the AST. For practical purposes, the segment corresponding to the preprocessing units is removed from the AST file that is present in this folder.

4.2.A separate node of the class type *UsingDirectiveDecl* is used to specify the namespaces that are globally used in the program. A special entry with the identifier is created for the namespace; in this case std.

4.3.Nodes corresponding to functions are created using the class *FunctionDecl*. The name of the function is stored in the symbol table. The return type and data types of the parameters are also identified within this node.

Children nodes from the class *ParamVarDecl* are used to declare the types and if an identifier is present for the parameter a new entry is made in the symbol table.

4.4.Block statements come under the class *CompoundStmt*. Nodes such as *DeclStmt, CXXOperatorCallExpr, IfStmt,* can all be units of the *CompoundStmt*. Each of them will be discussed shortly.

4.5.A Node of the Class *DeclStmt* normally consist of a basic rule such as the *VarDecl.* Such statements form a new entry in the symbol table, if not already present, else they simply refer to a previous entry id.

4.6.The *CXXOperatorCallExpr* for functionalities such as Cin and Cout are more complicated. Certain steps include referencing the methods from the ostream using *DeclRefExpr* rules. *ImplicitCastExpr* are used to refer to both the struct present in the ostream and to create a pointer to the string printed out as a *StringLiteral*, using an *ArrayToPointerDecay.*

4.7.Cascading Cout's are more interesting in the aspect that, all the calls to the ostream are made before the actual components of the cascading Cout are defined.

4.8.Function calls are made using the *FunctionToPointerDecay*. For functions that have been implicitly defined, when the actual function is parsed, the *FunctionDecl* specifies that the function name has been used already along with a reference to the old value.

4.9.Binary operators create 3 nodes, the operators as the parent, and the left and right chid as its children.

4.10. Variables are accessed using *ImplicitCastExpr* (The type) and *DeclRefExpr* (where the value is referenced from). Whereas, normal integers are declared using *IntegerLiteral*.

4.11. Return statements are created using the *ReturnStmt* class.

5. AST TRAVERSALS

Since classes are independent in the Clang AST (there is no base class), traversals are a huge issue. Even with the help of Glue classes and methods, a person would have to sit and make all the dynamic links with respect to the AST nodes himself. To get around this, Clang AST has a mechanism known as the RecursiveASTVisitor, which traverses only through the types that you care about, with a full knowledge of all the connections. However, contextual information about parent or children nodes cannot be obtained directly; hence we use ASTMatchers which uses expressions rather than types and give you a binding to your context.

In order to create a RecursiveASTVisitor based ASTFrontendActions we need to follow four basic steps:

5.1. Creating a FrontendAction:

It is the interface that allows user-specific actions to be executed and compiled. In this scenario, the user-specific action may be the ASTConsumer.

5.2. Creating an ASTConsumer

This is used to create different actions for the AST. There can be many different entry points for the AST, but we are concerned with only one, namely the ASTContext.

5.3. Using the RecursiveASTVisitor

The next step is to extract the relevant information from the AST. To do so we use the RecursiveASTVisitor as follows:

With the RecursiveASTVisitor we use a method VisitNodeType(NodeType *) to select certain nodes. This function returns a bool and works for most of the AST nodes, except certain location types such as TypeLoc (which are passed by value).

5.4. Accessing the SourceManager and ASTContext

In order to make sense out of the data present in certain nodes, we need to know its context. Context details (such as source locations and global identifiers) are not specifies in the node itself, but in the ASTContext and its associated source manager. Hence we need to incorporate the ASTContext into our RecursiveASTVisitor, which is available in the CompilerInstance when we make a call to the CreateASTConsumer.

6. ERROR MESSAGES IN LLVM

6.1.OVERVIEW OF ERRROR HANDLING IN LLVM

There are two types of errors that a LLVM handles fatal and non-fatal. Most of the non-fatal errors are handled through the LLVMContext itself, which manages the core global data of LLVM's infrastructure. Every method requires information regarding its context, which is what the LLVMContext provides.

The other types of errors, i.e. the fatal ones are handled by the ErrorHandling class. With he help of mutex's and assert, the program ensures that only one ErrorHandler is created. A mutex is also used to ensure that the ErrorHandler is deleted after use, without any interruptions from other segments of LLVM. There are several difference variances of the report_fatal-error function based on the data type of the reason parameter of the calling function. After checking for whether the error is reachable and then finally binding the data and reason to the corresponding dialogue, a map is created in a switch case to link the different types of errors to their respective conditions.

6.2. THE ASSERT MECHANISM

Assert is used constantly throughout the ASTContext class . It is used to ensure that all metadata's created are done so correctly. If any metadata kind is created illegally the assert function returns false and the procedure is halted, the error messages are informed using the DiagnosticHandler. Which is set in the setDiagnosticHandler function along with its context and filters. For example:

```
  // Create the 'nonnull' metadata kind.
unsigned NonNullID = getMDKindID("nonnull");
assert(NonNullID == MD_nonnull && "nonnull kind id drifted");
(void)NonNullID;
```

Errors are printed using the emitError function. A case is used to check and return the severity of the error.

The assert function comes from the <cassert> library. It checks for the condition. If the condition is no matched it returns the error message as specified in the function. The following section gives a couple of examples of error messaged form the ASTContext class, with respect to the creation of the metadata's.

6.3. EXAMPLES OF SPECIFIC ERROR MESSAGES

The following examples highlights the use of the assert method in the ASTContext and the error messages returned (highlighted in blue):

```
// Create the 'nonnull' metadata kind.
unsigned NonNullID = getMDKindID("nonnull");
assert(NonNullID == MD_nonnull && "nonnull kind id drifted”);
(void)NonNullID;

// Create the 'dereferenceable' metadata kind.
unsigned DereferenceableID = getMDKindID("dereferenceable");
assert(DereferenceableID == MD_dereferenceable &&
          "dereferenceable kind id drifted");
(void)DereferenceableID;

// Create the 'dereferenceable_or_null' metadata kind.
```

```
unsigned DereferenceableOrNullID =
getMDKindID("dereferenceable_or_null");
assert(DereferenceableOrNullID == MD_dereferenceable_or_null &&
       "dereferenceable_or_null kind id drifted");
(void)DereferenceableOrNullID;

// Create the 'make.implicit' metadata kind.
unsigned MakeImplicitID = getMDKindID("make.implicit");
assert(MakeImplicitID == MD_make_implicit &&
         "make.implicit kind id drifted");
(void)MakeImplicitID;

// Create the 'unpredictable' metadata kind.
unsigned UnpredictableID = getMDKindID("unpredictable");
assert(UnpredictableID == MD_unpredictable &&
        "unpredictable kind id drifted");
(void)UnpredictableID;
```

7. GENERAL OBSERVATIONS ABOUT LLVM IR

1. LLVM IR is not machine code, but sort of the step just above assembly.
2. LLVM IR does not differentiate between signed and unsigned integers.
3. Global symbols begin with an at sign (@).
4. Local symbols begin with a percent symbol (%).
5. All symbols must be declared or defined.

7.1 Mapping Basic Constructs to LLVM IR

7.1.1 Global variables:

```
int variable = 14;

int main()
{
    return variable;
}
```

Becomes:

```
@variable = global i32 14

define i32 @main() nounwind {
        %1 = load i32* @variable
        ret i32 %1
}
```

It's interesting to note that LLVM uses 'load' instruction when accessing the value of a global, variable; likewise store the value of the global variable using the 'store' instruction. This indicates that LLVM looks at global variables as pointers.

7.1.2 Constants

There seem to be different kinds of constants:

1. Constants that are inlined, and thus do not occupy allocated memory
2. Constants that are allocated memory

Example of type 1: %1 = add i32 %0, 17   (17 is an inlined constant)
Example of type 2: @hello = internal constant [6 x i8] c"hello \00"

A constant that HAS been allocated memory is in reality simply a global variable, whose visibility is limited by 'internal' (equivalent of 'private' in C++/Java)

7.1.3 Function Prototype

A function prototype is translated into an equivalent 'declare' declaration

```
int Bar(int value);
```

becomes:

```
declare i32 @Bar(i32)
```

7.1.4 Function Definitions

7.1.4.1 Simple Public Functions

```
int Bar(void)
{
        return 17;
}
```

becomes

```
define i32 @Bar() nounwind {
        ret i32 17
```

```
    }
```

7.1.4.2 Simple Private Functions

```
define private i32 @Bar() nounwind {
   ret i32 17
}
```

7. 1.5 Structures

```
struct Foo
{
   int a;
   char *b;
   double c;
};

int main(void)
{
   Foo foo;
   char **bptr = &foo.b;

   Foo bar[100];
   bar[17].c = 0.0;

   return 0;
}
```

becomes

```
%Foo = type {
   i32,        ; 0: a
   i8*,        ; 1: b
   double      ; 2: c
}

define i32 @main() nounwind {
   ; Foo foo
   %foo = alloca %Foo
   ; char **bptr = &foo.b
```

```
    %1 = getelementptr %Foo* %foo, i32 0, i32 1

    ; Foo bar[100]
    %bar = alloca %Foo, i32 100
    ; bar[17].c = 0.0
    %2 = getelementptr %Foo* %bar, i32 17, i32 2
    store double 0.0, double* %2

    ret i32 0
}
```

8. Name Mangling in Clang

Mangling used in Clang: (taken for LLVM article in AOSA by Chris Lattner)

```
 <encoding> ::- <name>
            ::- <local-name>
            ::- Z <function encoding> E <entity name>
```

```
 class A {
      void foo (int) {
        class B { };
     }
   };

   void foo () {
     class C {
       class D { };
void bar () {
  struct E {
    void baz() { }
  };
}
     };
   }
```

So the final result is "Z N 1A 3foo E i E 1B”

foo::C::D:

So the final result is "Z 3foo v E N 1C 1D E"

9 Optimization in Clang:

-O0: No optimization.
-O1: Get rid of dead code