Compilers Engineering 2019
Mini Assignment 1:
An Introduction to the LLVM Infrastructure, AST, IR and Compiler Options
Report

Sai Harsha Kottapalli
CS17BTECH11036

August 28, 2019

1 AST Structure

1.1 Hello world :

```
#include ⟨stdio.h⟩
int main() {
        printf("hello world\n");
        return 0;
}
```

To view the Clang-AST:
$ clang -Xclang -ast-dump -fsyntax-only hello.c

Details:

- **TranslationUnitDecl**
  We find that clang AST for any program starts with TranslationUnitDecl and all other nodes such as **TypedefDecl**, **RecordDecl**, **FieldDecl**, **ParmVarDecl** and **FunctionDecl** are children of the said node.

- **stdio.h**
  Clang also creates the AST for the header files included in the program. This can be easily verified to the fact that though the actual program is very short, the AST file generated has 681 lines out of which 670 belong to stdio.h or internal declaration of clang.

- **main**
  Refer Fig.1 for the AST which we need to analyze(The colour syntax makes analyzing easier so attaching picture is better than typing it out).
  FunctionDecl shows the function declaration main is of the type int(). The line number, column number help us to locate the occurence of the said types, identifiers etc for better analyzing\debugging.
  The pipes let us easily realte node to their parents and siblings.
  Also, we can notice the function references to previously declared ones also show the unique address. The entirety of the function definition is enclosed the **CompoundStmt** class.

- **printf**
  Through the AST we realize that "printf" actually is of the type int. This is required as it is probably being used to return the error code or success code.
  ImplicitCastExpr is helping us cast the value to int for the above use case. **StringLiteral** shows us that a character of array of exactly the size we require is made and passed as a const char * pointer reference.

- **return** The following **ReturnStmt** and **ReturnStmt** just shows 0 is returned at the end of function.

Figure 1: AST - 1st example

1.2 Addition :

```
#include ⟨stdio.h⟩
int add(int x, int y) {
        return x + y;
}

int main() {
        printf("%d", add(5, 4));
        return 0;
}
```

To view the Clang-AST:
$ clang -Xclang -ast-dump -fsyntax-only addition.c

Details: (Only differences wrt above program will be mentioned)

- **Heirarchy**
  Here, we notice that "add" and "main" functions are siblings in the tree as they should be.

- **func decl**
  As usual name in the function decl , the function name is shown, but this type since parameters are being passed, we see "int (int, int)" which means, the function returns "int" or is of the type int and requires (int, int) as parameters for it to function properly.

- **ParmVarDecl**
  We see this new keyword, which signifies the parameter variable declaration.

- **DeclRefExpr** To reference the memory address of the identifier.

- **BinaryOperator**
  As usual, function definition is encased in **CompoundStmt** but since this time instead of a **IntegralLiteral** we are returning a computed value, **BinaryOperator** comes into play.
  Since, we have declared function is of the type "int", we notice that

Binary Operator is of the type "int +" which is why all the **ImplicitCastExpr** are converting to "int" as a safety measure.
Also this is the case for array to pointer decay and function to pointer of function. Note: "LValueToRValue" is shown to signify that we are casting from identifier and that we have to fetch its value from the memory location. **IntegralLiterals** don't have this mentioned.
Note: We know that return statement is optional in c for "main", here in AST generation clang doesn't automatically added the **ReturnStmt**, so this must have been taken care even further inside the compiler chain.

Figure 2: AST - 2nd example

1.3 Binary Search(Iterative) :

```
int binarySearch(int arr[ ], int l, int r, int x) {
        while (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
                return mid;
        if (arr[mid] > x)
                r = mid - 1;
        else l = mid + 1;
        }
        return -1;
}
```

To view the Clang-AST:
$ clang -Xclang -ast-dump -fsyntax-only struct.c

Details: (Only differences wrt above program will be mentioned)

- **Keywords**
  We notice new keywords here, **WhileStmt**, **VarDecl**, **IfStmt**, **Unary-Operator** and **ArraySubscriptExpr**

- **WhileStmt** ⟨ line:2:3, line:9:3 ⟩
  Here, the line numbers tell us what is inside the while loop that has to be repeated. (Immediate child is the condition required other child is the loop.)

- **VarDecl**
  Declaring variables, shows name and type.

- **IfStmt**
  An if statement without "else" condition is similar to while loop, it has a **BinaryOperator** and expressions after that, if condition is evaluated to be true.
  From next if statement, we notice it has "has_else" which signifies this has an else condition associated with it.

- **UnaryOperator**
  Example, "-" or minus sign, on a number.

- **ArraySubscriptExpr**
  To reference a value from array.

  Note: For recursive method, the only main difference is the function is called using **DeclRefExpr** and the same memory address as the function is shown, similar to how we discussed in "printf" scenario in 1st example.

1.4 Some more programs.. :

Details: (Only differences wrt above program will be mentioned)

- **Keywords**
  We notice new keywords here (especially incase of struct), **RecordDeclStmt** (for struct name definition) and **FieldDecl** (for fields inside the struct).
  CallExpr (for calling functions) and **ForExpr for "for" loop**

2 AST Traversal

About:

The visitor mechanism to traverse the AST according to the link mentioned in the given pdf, allows us to traverse the nodes of the Clang AST in depth-first search way.
This can be verified using "$ clang-check hello.c -ast-dump" or with the help of the commands listed in the clang documentation (in the link shared in question).
From the output, we can infer that the traversal outputs the class-names/variable names with scope resolution with DFS as discussed. Basically, what this means is that of the three core classes of clang, all the "decl"(declaration) classes w.r.t. clang are printed.

How to:

- **FrontendAction**
  When we provide the required actions, this is responsible for executing those during compilation.
  The structure is an interface and is predefined , so we only have to follow it and implement **CreateASTConsumer** under it.

- **ASTConsumer**
  Though this interface has many entry points to start with, since, as of now we only deal with the translation unit. Clang documentation recommends the use of only "HandleTranslationUnit" for all of our usage.

- **RecursiveASTVisitor**
  This is the main interface which is responsible for traversal.
  Here, we can give the required conditions(by grabbing required details, Ex: **getQualifiedNameAsString()** for name) to find nodes that we require and then accordingly mark them(using dump() function).
  We also have the option to continue doing so(return true) or stop when necessary.(return false).
  If we dump everything by default, we get all the declaration names as it should, because we call dump() (therefore marking it) for all declarations.

- **ASTContext**
  On the rare cases where we must use global variable, this information of course wont be present in the nodes.
  Clang documentation therefore asks us to use **ASTContext** for this scenario.

3 Error Messages for Handling

- **Where to find:**
  When we lookup the directory hierarchy in the documentation, it is mentioned about "lib/Support" folder. Upon inspecting we find "ErrorHandling.cpp" inside it.

- **Assert**
  Firstly, to use assert, we need to include ¡cassert¿ header file.
  This is really helpful in checking the pre-conditions or assumptions that we have (especially before a loop or function definition).
  The main advantage is that debugging the code becomes really easy when compared to the code without assert. But there is a trade-off. Using "assert" will increase the program execution time, so it is mostly used as mentioned above.
  Ex:
  assert(!ErrorHandler && "Error handler already registered!\n");
  assert(!ErrorHandler && "Bad alloc error handler already registered!\n");
  assert(old == nullptr && "new-handler already installed");
  which are accordingly called w.r.t. later execution using those variables. Here, we check if there is any error handler being referenced or is null, and accordingly the associated error message is printed.
  In the docs, it is mentioned that for release build asserts are disabled. This makes sense as there are many reasons.
  1. More errors/ warnings (referencing something that is not present in user's code).
  2. User needs to know about associated header files and accordingly install them when it is not really required for most cases.
  3. Program Behavior is undefined when assertion results in false.
  Although for worst case scenario if user does trigger some buggy code, "llvm_unreachable" takes care of it by either exiting the program or skipping that particular branch of the code.
  References: https://llvm.org/docs/CodingStandards.html

```
assert(Ty->isPointerType() && "Can't allocate a non-pointer type!");

assert((Opcode == Shl || Opcode == Shr) && "ShiftInst Opcode invalid!");

assert(idx < getNumSuccessors() && "Successor # out of range!");

assert(V1.getType() == V2.getType() && "Constant types must be identical!");

assert(isa<PHINode>(Succ->front()) && "Only works on PHId BBs!");
```

Figure 3: More examples for assert function

4 LLVM IR

- Sum of two arbitrarily large numbers (no limit no size of the two numbers).
- Set implementation
- Generating BST from input pre-order traversal.
- Binary search iterative
- Finding BFS and shortest path between two nodes
- Findings:
  It is basically a target independent assembly like language which acts as an intermediate representation which an advantage of infinite registers.
  It is strongly typed.
  The generated LLVM IR is attached as .ll files.
  To generate llvm ir from a source code:
  $ clang -S -emit-llvm hello.c

  source_filename gives us the file name and generally ModuleID is same as that.
  target datalayout is responsible for showing the details of the target system's supported memory layout.
  [5 x $i32$] implies it is an array of size 5 and of type i32. similarly for n-dimensional arrays.
  Every function specifies the return type an "(..)" which shows the type of parameters accepted by it.
  Each function has comments (starts with ;) which specifies the function attributes.
  %5, %6 , etc are register names which are allocated a type using "alloca" and aligned for proper retrieval during memory accesses.
  We use store keyword when operating and storing data in memory.
  Most of the ll files is divided into basic blocks(which have no jumps in between them) and can be connected via CFG.
  A label is present before each of these basic blocks for identifying which block has to be accessed next during the execution flow(jumps are made with the help of "br").

Therefore, Label's use is to branch out.
"load" is to get data from main memory and store it in register.
"preds" tell us which basic block might call the current basic block. llvm also has struct data type which stores the data types declared inside it.
All functions referenced via header files are defined using "declare" keyword in ll file.
Phi node is special instruction which chooses a value based on it's pred during the flow.
"sext" is used for type casting but both values should be of the integer class/family.
"call" is used to call a function with its type and name.
Documentation also mentions that IR is static single assignment which means that all variables are declared before use and allocated only once.

5 Assembly language:

- Name Mangling
  It can be the case that, when linker is trying link programs in a language, there are functions with same names. In these cases, there would be a conflict.
  Hence, as a precaution a unique name is chosen for each function, and these unique names are called name mangling.
  Example: When we declare a variable in different scope or when we overload + symbol for string addition as well as mathematic addition, etc.
  For the trivial program, there is a "_Z1" prepended.
  When compared with other programs, we can infer that "_Z" is prepended as a convention and the number signifies the scope of the variable.
  For classes, it prepended "_ZN"
  Examples: On redeclaring functions: _Z4def1v and _Z4def2i.

6 Compiler toolchain and options:

compiling cpp files with clang - $ clang++ -Wall -std=c++11 test.cc -o test.
-Wall prints warning messages while compiling.
llvm-ar: archiving llvm bitcode files.
llvm-as: assembler for transformation of human readable LLVM assembly to LLVM bitcode.
llvm-dis: opposite to llvm-as, bitcode to human readable assembly.
llvm-link: linking multiple llvm modules
lli: llvm interpreter for bitcode
llc: bitcode to native assembly
opt: Apply transformations on bitcode to generate a optimized verion of the input bitcode.(can pass different levels -O1,2,3,4).
Here, -O0 is no optimization(fastest) and easier to debug. from 1,2,3 the time taken increases while the optimization level applied also increases.
We notice though optimized larger code is generated some times.
-std= : to provide a standard for compiling in that particular standard.
-arch for providing architechture.
-fsyntax-only : just check correctness
For most of the programs, the number of resultant basic blocks decreased when optimization was applied. llvm-diff for differences in files. Example when compared for .ll files, it shows where the functions existed (only left module or only right module).

References: https://llvm.org/docs/GettingStarted.html

7 Kaleidoscope

Only double type is supported(advantage being types need not be passed).
arrays, structs, vectors, etc are supported.
global variables are supported but need to make an instance of GlobalVariable class.
Supports Accurate Garbage Collection.
debugger support is present
Docs claim - object orientation, generics, database access, complex numbers, geometric programming.
Target Independence.

It has an IRBuilder class which tracks the position to insert instructions.

References: http://llvm.org/docs/tutorial/index.html